

# Ausführungsumgebung für C--

Günther Blaschek  
V1.0, 2014-06-18

## Allgemeines

Die Ausführungsumgebung soll zumindest folgende Funktionen bieten:

- Editieren, Speichern und Laden von Programmen
- Übersetzung durch den Compiler
- Anzeige von Fehlern und Fehlerstellen
- Festlegen von Eingabedaten für das Programm
- Anzeige von Programm-Ergebnissen (Ausgaben mit *print*)
- Starten, Unterbrechen und Abbrechen von Programmen
- Beobachtung des Programmablaufs (Debugging)

Die Ausführungsumgebung ist die Hauptfunktion, die alle anderen Komponenten steuert. Sie sorgt dafür, dass bei Änderungen am Programmtext der Compiler aufgerufen wird, hilft bei der Lokalisierung von Programmfehlern, startet den Interpreter und zeigt Informationen über den Programmablauf. Wenn der Interpreter läuft, meldet er den Fortschritt mit Hilfe einer Callback-Funktion an die Ausführungsumgebung, die daraufhin die angezeigten Informationen aktualisieren sowie die Ausführung unterbrechen oder abbrechen kann.

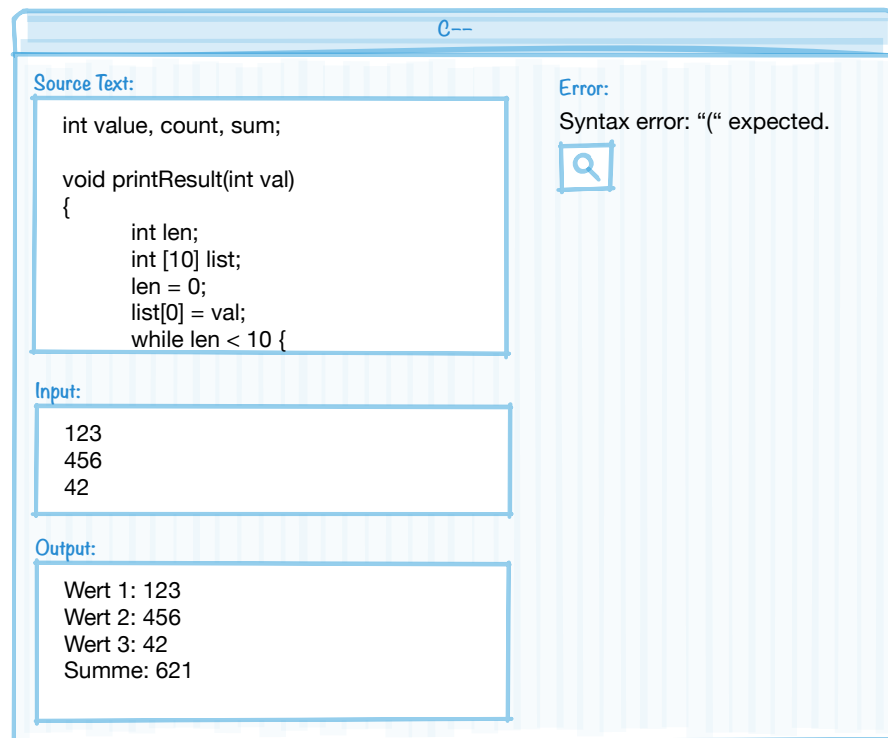
Der Einfachheit halber ist C-- so definiert, dass während der Programmausführung keine Benutzerinteraktion erfolgt. Die Eingabedaten für das Programm werden vorweg in ein Textfeld geschrieben und von dort ohne weitere Benutzereingriffe vom Programm gelesen.

Die Bedienung des Systems erfolgt in einem einzigen Fenster, das alle wesentlichen Informationen zeigt. Der Einfachheit halber ist nicht vorgesehen, mehrere Programme in gleichzeitig geöffneten Fenstern zu bearbeiten (oder gar laufen zu lassen).

## Bildschirmaufbau

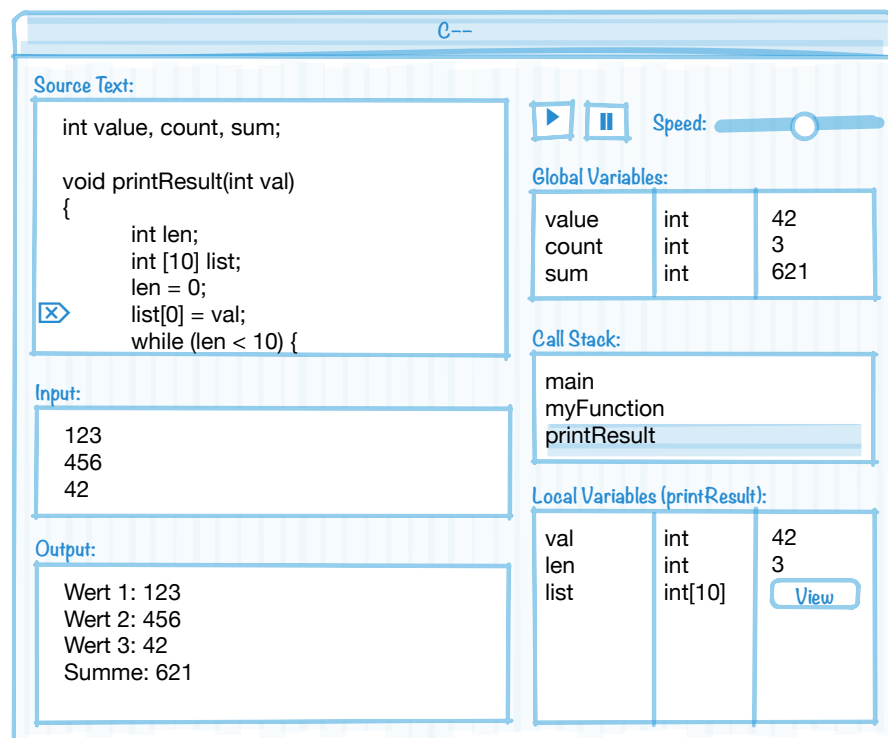
Das Fenster zum Bearbeiten und Ausführen von Programmen ist in zwei Teile geteilt: Links steht das Feld mit dem Programmtext, darunter der Eingabetext und die vom Programm gelieferte Ausgabe. Der Programmtext und die Eingabe können editiert werden; das Outputfeld wird beim Programmlauf gefüllt und bei jedem Programmstart automatisch geleert.

Der Programmtext und die Eingabe werden gemeinsam in einer Datei gespeichert. Wenn die Ausführungsumgebung beendet und neu gestartet wird, können diese Teile wieder geladen werden.



Wenn das Programm Fehler enthält, werden im rechten Teil des Fensters entsprechende Informationen angezeigt (siehe folgenden Abschnitt *Behandlung von statischen Programmfehlern*).

Wenn das Programm korrekt ist, werden statt dessen im rechten Teil des Fensters Elemente zum Steuern der Programmausführung und zum Inspizieren des Programmszustands angezeigt:



## Behandlung von statischen Programmfehlern

Wenn das Programm editiert wird, wird nach jeder Änderung sofort versucht, den Quelltext zu übersetzen. Wenn der Compiler einen Fehler feststellt, wird der erste Fehler im rechten Teil des Fensters angezeigt. Die Schaltfläche mit der Lupe hilft beim Finden der Fehlerstelle, indem die Einfügemarke an die entsprechende Stelle im Quelltext gesetzt wird.

Der Compiler liefert eine Liste von Fehlern. Meist führt aber der erste Fehler zu Folgefehlern, so dass durch Korrektur des ersten Fehlers etliche weitere Fehler verschwinden. In der ersten Ausbaustufe wird daher nur der erste Fehler angezeigt.

Wenn sich herausstellen sollte, dass die probeweise Übersetzung nach jeder Änderung für eine sofortige Aktualisierung zu langsam ist oder zu lästigem Flackern führt, gibt es zwei Lösungsmöglichkeiten:

- Explizites Starten des Compilers. Solange das Programm noch nicht übersetzt wurde oder es seit der letzten Übersetzung geändert wurde, bleibt die rechte Fensterhälfte bis auf einen "Compile"-Button leer. Durch Drücken des Buttons (alternativ durch einen Tastendruck) wird die Übersetzung ausgelöst.
- Übersetzen nur in längeren Eingabepausen. Laufende Übersetzungen werden während der Texteingabe unterdrückt. Der Compiler wird automatisch gestartet, wenn längere Zeit (z.B. zwei Sekunden lang) keine Taste gedrückt wurde.

## Starten, Unterbrechen und Anhalten von Programmen

Mit dem Run-Knopf (▶) wird die Ausführung gestartet. Wenn das Programm läuft, erscheinen die Ergebnisse von *print*-Anweisungen im Output-Feld, während die *read*-Funktion Zeichen aus dem Input-Feld liest.

Man kann das Programm einfach laufen lassen. Es hält dann nach der Ausführung der letzten Anweisung an. Wenn das Programm läuft, wird der Pause-Knopf (⏏) aktiv, so dass man das Programm jederzeit unterbrechen kann.

Wenn das Programm unterbrochen wurde, wird der Pause-Knopf zu einem Stopp-Knopf (■):



Man hat nun zwei Möglichkeiten: mit Run die Ausführung wieder aufnehmen oder mit Stopp das Programm abbrechen.

Wenn ein Programm gestartet wird, läuft der Interpreter in einem eigenen Thread. *print*- und *read*-Aufrufe führen zu Aufrufen von Callback-Funktionen der Ausführungsumgebung, die darauf das jeweils nächste Zeichen an das Output-Feld anfügt oder ein neues Zeichen aus dem Input-Feld liefert. Beim Lesen von Zeichen kann der Fortschritt im Input-Feld visualisiert werden, beispielsweise durch blassere Darstellung der bereits konsumierten Zeichen:

Input:

  
123  
456  
42

Weiters meldet der Interpreter mit einer eigenen Callback-Funktion den Fortschritt. Das geschieht typischerweise nach jeder Anweisung. Der Interpreter gibt bei diesem Callback u.a. die aktuelle Anweisungsnummer mit.

Die Ausführungsumgebung kann darauf auf verschiedene Arten reagieren. Sie kann insbesondere den Fortschritt im Quelltext-Fenster durch Hervorheben der aktuellen Anweisung anzeigen und die Werte von Variablen aktualisieren.

Der Callback läuft im Interpreter-Thread. Die Ausführungsumgebung kann mit *suspend* die Ausführung anhalten und später mit *resume* fortsetzen. Das kann insbesondere für die Pause-Funktion genutzt werden: Wenn der Benutzer während der Ausführung die Pause-Taste betätigt, wird das vorerst nur von der Ausführungsumgebung gemerkt. Beim nächsten Fortschritts-Callback wird dann der Interpreter angehalten. Durch erneutes Drücken der Run-Taste wird der Interpreter wieder fortgesetzt.

Die Ausführungsumgebung kann auch die Ausführung abbremesen. Der Interpreter wird blockiert, solange der Fortschritts-Callback die Kontrolle nicht wieder an den Interpreter zurückgegeben hat. Die Ausführungsumgebung kann daher gezielt Verzögerungen einbauen, um die Ausführung langsamer zu machen. Die Ausführungsgeschwindigkeit kann dabei durch den "Speed"-Regler eingestellt werden.

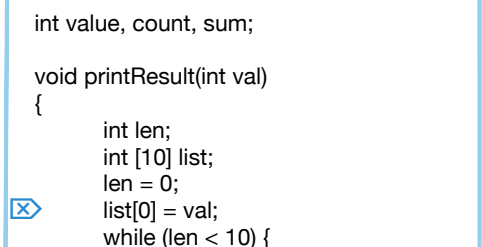
*Hinweis:* Wenn die Geschwindigkeit auf "volle Kraft voraus" eingestellt ist, sollte die Ausführung so schnell wie möglich erfolgen. Es sollte also nichts gebremst und auch kein Fortschritt angezeigt werden, insbesondere keine Hervorhebung der aktuellen Zeile und keine Aktualisierung von Variablenwerten. Eine Aktualisierung erfolgt dann nur beim Drücken der Pause-Taste.

## Breakpoints

Zur Fehlersuche soll es möglich sein, an beliebigen Stellen im Programm Breakpoints zu setzen. Wenn das Programm eine Stelle mit einem Breakpoint erreicht, wird die Ausführung angehalten.

Das Setzen und Entfernen von Breakpoints kann z.B. mit einem Kontextmenü (Rechts-Klick) erfolgen. Anweisungen mit Breakpoints sollen durch ein Symbol im Quelltext angezeigt werden, z.B.:

Source Text:



```
int value, count, sum;

void printResult(int val)
{
    int len;
    int [10] list;
    len = 0;
    list[0] = val;
    while (len < 10) {
```

Die Liste der Breakpoints wird von der Ausführungsumgebung verwaltet. Wenn der Interpreter eine Fortschrittsmeldung mit einer Zeile liefert, die einen Breakpoint enthält, wird die Ausführung angehalten.

Die Breakpoints werden automatisch gelöscht, wenn der Quelltext geändert wird.

Wenn während der Ausführung ein Fehler auftritt (z.B. Indexüberlauf oder Division durch 0), soll das Programm an der Fehlerstelle angehalten werden, als ob dort ein Breakpoint gesetzt wäre. Der Zustand des Programms kann dann untersucht werden; ein Fortsetzen mit dem Run-Knopf hat dann allerdings keinen Sinn und soll verboten werden.

Breakpoints beziehen sich auf Quelltextzeilen und sind daher nur ungenaue Angaben. Das ist beispielsweise der Fall, wenn mehrere Anweisungen in einer Zeile stehen. Der Interpreter meldet in solchen Fällen mehrmals nacheinander den Beginn einer Anweisung mit der selben Zeilennummer. Das ist insbesondere zu beachten, wenn nach einem Breakpoint die Ausführung fortgesetzt wird. Wenn der Interpreter die nächste Fortschrittsmeldung liefert, sollen Anweisungen mit der gleichen Zeilennummer nicht sofort wieder zu einer Unterbrechung führen. Der Breakpoint wird erst wieder "scharf", wenn in der Zwischenzeit eine Anweisung mit einer anderen Zeilennummer ausgeführt wurde. Anders ausgedrückt: Die Ausführung wird in der Zeile x angehalten, wenn in der Zeile x ein Breakpoint gesetzt ist, der Interpreter meldet, dass er in der Zeile x angekommen ist, und die vorherige Meldung sich auf eine *andere* Zeile bezog.

## Laufzeitfehler

Zur Laufzeit können Fehler auftreten, die eine weitere Ausführung unmöglich machen. Das sind z.B. Indexfehler beim Zugriff auf Array-Felder, Division durch 0 oder Stack-Überlauf durch zu tiefe Rekursion.

Wenn der Interpreter einen Laufzeitfehler feststellt, meldet er das mit einem Callback wie bei einer Fortschrittsmeldung. Die dabei mitgegebenen Datenstrukturen sind die selben (mit einem zusätzlichen Text, der die Fehlerursache beschreibt), so dass die Ausführungsumgebung die Fehlerstelle im Quelltext sowie die Variablenwerte anzeigen kann. Die Ausführungsumgebung blockiert den Interpreter und verbietet das Fortsetzen der Ausführung mit dem Run-Button.

Die Fehlerursache kann im Fenster rechts oben angezeigt werden. Der Speed-Regler und der Run-Button haben in dieser Situation keine Bedeutung und können daher entfallen, z.B. so:



## Inspektion von Variablen

Es gibt zwei Arten von Variablen: Globale Variablen werden in einer eigenen Liste angezeigt und sind immer sichtbar. Lokale Variablen gehören zu Prozeduren; sie existieren nur, solange die umschließende Prozedur läuft.

Welche Prozeduren aktiv sind, wird in einer "Call Stack"-Liste angezeigt. Durch Auswählen einer Prozedur in dieser Liste werden unter "Local Variables" die zu dieser Prozedur gehörenden Variablen angezeigt.

Die Variablenlisten sind dreispaltig angeordnet. Die erste Spalte enthält den Namen, die zweite den Datentyp und die dritte den Wert der Variablen.

Elementare Werte (mit den Typen *float*, *int* und *char*) werden direkt in der Liste angezeigt. Strukturierte Variablen (Arrays und Structs) haben in der Wertspalte eine Schaltfläche, mit der man in den Inhalt "einsteigen" kann. Die Variablenliste wird dann durch eine Detailansicht der Variablen ersetzt. Für ein Array werden dann z.B. die darin enthaltenen Werte angezeigt:

Array list [10]:

[0]	int	42	←
[1]	int	6	
[2]	int	7	
[3]	int	0	
[4]	int	0	
[5]	int	0	
[6]	int	0	

Bei einem Struct werden statt der Indizes die Namen der Felder angezeigt.

Mit der Schaltfläche "←" kann man aus einer strukturierten Variablen wieder "aussteigen" und zur übergeordneten Variablenansicht zurückkehren.

Structs und Arrays können auch geschachtelt sein. Es ist also möglich, dass ein Struct ein Array enthält, dessen Elemente wiederum Structs sind. Das Ein- und Aussteigen in die Details kann daher auch über mehrere Stufen erfolgen.

Wenn das Programm fortgesetzt wird, soll die Variablenansicht so weit wie möglich erhalten bleiben. Die Ausführungsumgebung soll also überprüfen, ob z.B. nach der Ausführung eines Schritts oder beim nächsten Breakpoint die gerade angezeigte Prozedur noch "lebt". Wenn das der Fall ist, sollen die neuen Variablenwerte angezeigt werden. Wenn die gerade angezeigte Prozedur jedoch beendet wurde, soll im Call Stack bis zur nächsten noch lebenden Prozedur zurückgegangen werden.

## Ausbaumöglichkeiten

Die Ausführungsumgebung kann vom hier spezifizierten Umfang ausgehend nahezu beliebig erweitert und komfortabler gemacht werden. Mögliche Erweiterungen sind:

- Anzeige aller vom Compiler gefundenen Fehler (statt nur des ersten)
- "Breadcrumb Trail" bei der Inspektion von strukturierten Variablen, z.B. in der Form `obj.list[5].elem`, so dass man den Weg zur aktuellen Variablen sieht und mit einem Klick zu einem beliebigen übergeordneten Element zurückkehren kann.
- Anzeige strukturierter Variablen in "ausklappbaren" hierarchischen Listen (mit "+"- und "-"-Knöpfen)
- Ein- und Ausschalten von Breakpoints durch Klicken in der linken Randspalte im Quelltext.
- Syntax-Coloring des Quelltextes