Lernziele:
- Umgang mit einfach verketteten Listen
- Vertiefung der Programmierkenntnisse

Vorbereitung: Compiler aktualisieren

1. Wechselt in das Verzeichnis **abc-llvm** mit dem Source-Code für den Compiler.
2. Führt das Kommando **git pull** aus.
3. Führt das Kommando **make** aus.
4. Führt das Kommando **sudo make install** aus.

Schritt 1

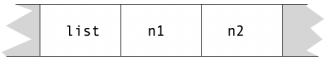
Folgendes Programm **list.abc** soll verwendet werden, um die Grundidee einer einfach verketteten Liste (Singly-Linked-List) zu erkunden:

```
1  @ <stdio.hdr>
2  @ <stdlib.hdr>
3
4  struct Node
5  {
6      val: int;
7      next: -> Node;
8  };
9
10 fn main()
11 {
12     local list: -> Node = nullptr;
13
14     // prepend new node to list
15     local n1: -> Node = malloc(sizeof(*n1));
16     n1->val = 42;
17     n1->next = list;
18     list = n1;
19
20     // prepend new node to list
21     local n2: -> Node = malloc(sizeof(*n2));
22     n2->val = 12;
23     n2->next = list;
24     list = n2;
25
26     printf("list->val = %d\n", list->val);
27     printf("list->next->val = %d\n", list->next->val);
28 }
```

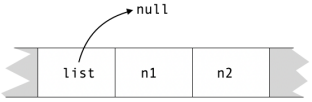
Aufgaben:

1. Bevor wir uns das Programm genauer anschauen, sollte es jeder abtippen, übersetzen und ausführen.
2. Im nächsten Schritt sollt ihr nachvollziehen, was in der Funktion **main** nach jeder Anweisung passiert. Insbesondere sollt ihr beschreiben, wie sich nach jeder Anweisung verschiedene Speicherflächen ändern. Für die ersten Anweisungen ist eine mögliche Beschreibung vorgegeben. Für die weiteren Anweisungen sollt ihr in ähnlicher Weise den Effekt einer Anweisung beschreiben.

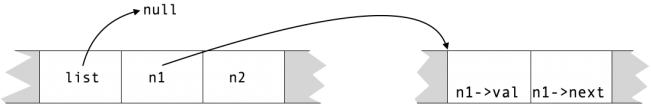
3. In main werden die drei Variablen **list**, **n1**, **n2** definiert. Es ist dabei absolut irrelevant, an welcher Stelle die lokalen Variablen definiert werden. Der Compiler kann erkennen, wie viele lokale Variablen insgesamt in einer Funktion verwendet werden. Der Speicherplatz kann deshalb bereits beim Funktionseintritt für alle lokalen Variablen auf dem Stack reserviert werden. Diese Speicherflächen sind dann natürlich uninitialisiert.



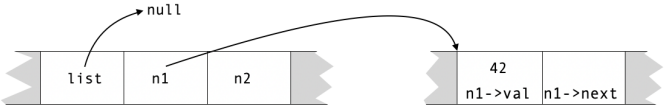
In Zeile 12 wird **list** zur Laufzeit mit einem Null-Zeiger ("ungültiger Zeiger") initialisiert. Und diesen Effekt kann man wie folgt visualisieren:



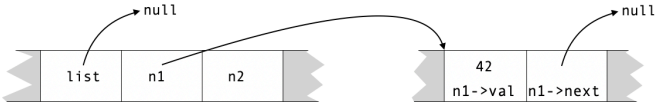
In Zeile 15 wird auf dem Heap ein Speicherblock der Größe **sizeof(*n1)** (was hier äquivalent zu **sizeof(Node)** ist) allokiert und **n1** mit der Adresse des allokierten Speicherblocks initialisiert:



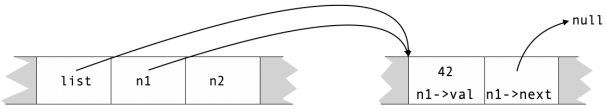
Dieser Speicherblock wird als neuer Listenknoten verwendet. In Zeile 16 wird in diesem Knoten das Member **val** mit 42 initialisiert:



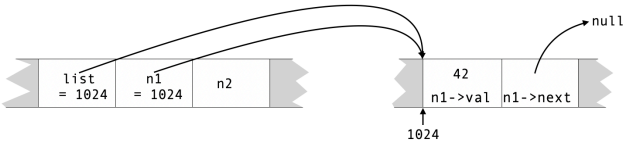
In Zeile 17 wird im Knoten das Member **next** mit dem Wert von list initialisiert:



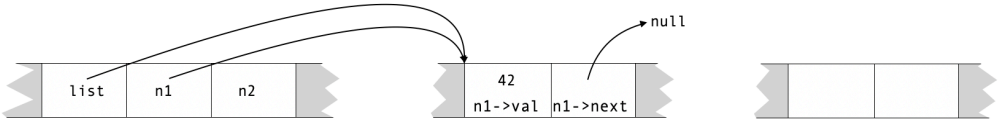
In Zeile 18 wird **list** auf die Adresse des Knotens gesetzt:



Es ist auch hilfreich, für Speicherflächen konkrete (aber ausgedachte) Adressen zu verwenden, um sich nochmals klarzumachen, dass eine Zeigervariable eigentlich nur eine Adresse speichert. Hat der mit **malloc** allokierte Block zum Beispiel die Adresse 1024 und wird für **nullptr** auf der Architektur tatsächlich der Wert 0 verwendet, dann kann man die Speicherflächen nach Zeile 18 auch so beschreiben:



Wir werden später sagen, dass in Zeile 12 eine leere Liste angelegt wurde und in den Zeilen 15 bis 18 ein neuer (Listen-)Knoten mit dem Wert 42 an den Anfang der Liste angehängt wurde. Das Prinzip "einen neuen Knoten an den Anfang anhängen" wird aber erst klar, wenn ihr nun analog den Effekt der Zeilen 21 bis 24 beschreibt. Ihr könnt dazu schrittweise nachfolgende Vorlage verwenden:



Durch die Beschreibung der Speicherflächen sollte dann erkennbar sein, dass eine Liste mit zwei Knoten angelegt wurde, wobei der erste Knoten den Wert 12 und der zweite den Wert 42 enthält. Dass ein Knoten der letzte in der Liste ist, wird dadurch festgelegt, dass das Member **next** ein Null-Zeiger ist.

4. Ersetzt den Code in den Zeilen 26 und 27 durch folgende For-Schleife:

```
printf("list:\n");
for (local n: -> Node = list; n != nullptr; n = n->next) {
    printf("n->val = %d\n", n->val);
}
```

   (A) Bestätigt, dass das Programm die gleiche Ausgabe wie zuvor produziert.
   (B) Beschreibt den Kontrollfluss der Schleife, falls notwendig, mit einem Flowchart, um eurem Nebensitzer (oder mir in der Prüfung) zu erklären, welche Werte **n** in der Schleife annimmt und wann die Schleife abbricht.
   (C) Löscht im Programm die Zeilen, in denen der Knoten **n1** an den Anfang der Liste hinzugefügt wird (im Programm oben sind das die Zeilen 14 bis 18). Bestätigt, dass die Liste jetzt nur noch einen Knoten mit dem Wert 12 enthält und diese Liste ausgegeben wird.
   (D) Löscht im Programm die Zeilen, in denen der Knoten **n2** an den Anfang der Liste hinzugefügt wird (im Programm oben sind das die Zeilen 20 bis 24). Bestätigt, dass die Liste jetzt "leer" ist (d.h. keinen Knoten enthält) und die Ausgabe auf keine ungültigen Speicherflächen zugreift, sondern einfach nichts ausgibt.
   (E) In C/C++/ABC ist es üblich, dass man einen Zeiger in einer Bedingung nicht explizit mit nullptr vergleicht, sondern ausdrückt "Zeiger ist gültig" oder "Zeiger ist ungültig". Ändert die Zeilen für die Ausgabe deshalb ab in:

```
for (local n: -> Node = list; n; n = n->next) {
    printf("n->val = %d\n", n->val);
}
```

5. In die Liste sollen jetzt wieder Knoten eingefügt werden. Diesmal werden wir dafür aber eine Schleife verwenden. Fügt nach der Initialisierung von list folgende Zeilen ein:

```
for (local i: int = 1; i <= 3; ++i) {
    local n: -> Node = malloc(sizeof(*n));
    n->val = i;
    n->next = list;
    list = n;
}
```

   Skizziert wieder die relevanten Speicherflächen beim Ausführen des Programms, um eurem Nebensitzer (oder mir in der Prüfung) zu erklären, wie neue Knoten in die Liste hinzugefügt werden. Ändert den Code so, dass 10 Knoten eingefügt werden.
6. Die Ausgabe einer Liste kann man in diese Funktion **printList** auslagern:

```
fn printList(list: -> Node)
{
    for (; list; list = list->next) {
        printf("val = %d\n", list->val);
    }
}
```

   Verwendet diese Funktion in **main**, um die Liste auszugeben. In dieser Funktion wird die Variable **list** verändert. Erklärt eurem Nachbarn (oder mir in der Prüfung), wieso die Variable **list** in **main** nach dem Aufruf sich nicht verändert hat.
7. Im letzten Schritt soll nun auch der Code für das Hinzufügen eines Knotens in eine Funktion **prependToList** ausgelagert werden. Mein Versuch im nachfolgenden Programm ist dabei aber leider nicht mit Erfolg gekrönt. Bestätigt zunächst, dass sich das Programm zwar übersetzen und ausführen lässt, aber in Zeile 34 eine leere Liste ausgegeben wird.

```
 1 @ <stdio.hdr>
 2 @ <stdlib.hdr>
 3
 4 struct Node
 5 {
 6     val: int;
 7     next: -> Node;
 8 };
 9
10 fn prependToList(list: -> Node, val: int)
11 {
12     local n: -> Node = malloc(sizeof(*n));
13     n->val = val;
14     n->next = list;
15     list = n;
16 }
17
18 fn printList(list: -> Node)
19 {
20     for (; list; list = list->next) {
21         printf("val = %d\n", list->val);
22     }
23 }
24
25 fn main()
26 {
27     local list: -> Node = nullptr;
28
29     for (local i: int = 1; i <= 10; ++i) {
30         prependToList(list, i);
31     }
32
33     printf("list:\n");
34     printList(list);
35 }
```

Offensichtlich hat hier die Call-by-Value-Falle zugeschlagen. Beim Hinzufügen eines neuen Knotens an den Anfang einer Liste ändert sich natürlich der Zeiger auf das erste Element. Da in Zeile 30 aber eine Kopie von list an prependToList übergeben wird, kann sich list durch den Aufruf nicht ändern. Man muss also die Adresse von list übergeben und den Aufruf ändern in

```
prependToList(&list, i);
```

Als Konsequenz muss nun aber auch die Implementierung der Funktion angepasst werden. Und das ist nun eure Aufgabe.

Schritt 2

Eine Funktion zur Ausgabe einer Liste kann man auch rekursiv implementieren:

```
fn printListRecurive(list: -> Node)
{
    if (list) {
        printf("val = %d\n", list->val);
        printListRec(list->next);
    }
}
```

Verwendet diese Funktion in **main**, um die Liste ein zweites Mal auszugeben:

```
printf("list:\n");
printList(list);

printf("list:\n");
printListRecurive(list);
```

Aufgaben:

1. Bestätigt, dass beide Aufrufe die gleiche Ausgabe erzeugen.
2. Erklärt, wie bei einer Liste mit drei Knoten der Stack beim Aufruf von **printListRecursive** verwendet wird.
3. Schreibt eine Funktion **printListReverse**, die den Inhalt einer Liste in umgekehrter Reihenfolge ausgibt und verwendet diese Funktion in main, um die Liste in umgekehrter Reihenfolge auszugeben.

Schritt 3

Bei der Verwendung von dynamischem Speicher sind zwei Dinge von wesentlicher Bedeutung und wurden bisher nicht beachtet:

1. Der Aufruf von **malloc** kann fehlschlagen, und in diesem Fall ist der Rückgabewert ein Nullzeiger.
2. Speicher, der mit **malloc** allokiert wurde, sollte mit free wieder freigegeben werden.

In unserem Programm wird **malloc** nur in **prependToList** verwendet. Hier kann man im Fall, dass kein Speicher mehr verfügbar ist, das Programm mit einer Fehlermeldung und Exit-Code 1 beenden:

```
10 fn prependToList(list: -> -> Node, val: int)
11 {
12     local n: -> Node = malloc(sizeof(*n));
13     if (!n) {
14         printf("out of memory\n");
15         exit(1);
16     }
17     n->val = val;
18     n->next = *list;
19     *list = n;
20 }
```

Ansonsten würde das Programm mit einem Segmentation-Fault terminieren, wenn **malloc** einen Nullzeiger geliefert hat und dieser beim Dereferenzieren in Zeile 17 verwendet wird.

Warum ist das besser?

1. Ein Segmentation-Fault ist die schlimmste Fehlermeldung überhaupt. Es bedeutet, dass es irgendwo ein Problem mit Zeigern gibt und das Betriebssystem teilt einem mit erhobenem Mittelfinger mit: "Viel Spaß bei der Fehlersuche".
2. Durch die Ausgabe einer Fehlermeldung weiß man, was schief und wo etwas schief ging. Man kann hier auch ein **assert** verwenden, dann wird bei der Fehlermeldung auch der Name der Source-Datei und die Zeilennummer ausgegeben.

Etwas mehr Denkarbeit ist bei der Freigabe der Listenknoten erforderlich. Hier eine Funktion zur Freigabe einer Liste, die undefiniertes Verhalten hat (was früher oder später zu einem Segmentation-Fault 👆 führt):

```
fn freeList(list: -> Node)
{
    for (; list; list = list->next) {
        free(list);
    }
}
```

Aufgaben:

1. Verwendet diese Funktion zu Testzwecken dennoch und ruft die Funktion am Ende von main auf. Bei mir hat es zumindest heute zu keiner Fehlermeldung geführt, aber das ist das Tückische an Programmen mit undefiniertem Verhalten.
2. Der Grund für das undefinierte Verhalten ist, dass mit free der Knoten am Ende des Zeigers von **free** freigegeben wird und dann auf **list->next** zugegriffen wird.

   Etwas mehr Hintergrund: In der Regel wird eine mit free freigegebene Speicherfläche nicht verändert, das hängt aber von der Implementierung von free ab (und die ist plattformabhängig). Wenn man also kein Programm mit Multithreading hat, kann scheinbar alles gut gehen. Wenn man aber Multithreading hat, kann es sein, dass nach dem **free(list)** ein anderer Thread die freigegebene Speicherfläche allokiert und verändert, bevor man auf **list->next** zugreift.

   Nun zur eigentlichen Aufgabe: Schreibt eine Funktion freeList, die ordentlich arbeitet, indem der Speicher eines Knotens erst dann freigegeben wird, wenn der Speicher des Nachfolgeknotens freigegeben wurde.