Lernziele:

- Umgang mit Pointern in ABC.
- Typumwandlungen.

Schritt 1: Pointer und Pointer auf Pointer

Hier ist das Programm **ex1.abc** mit einigen Kommentaren und Anweisungen:

```
1  @ <stdio.hdr>
2
3  fn main()
4  {
5      local
6          a: int = 42,
7          b: int = 123,
8          p: -> int = &a,
9          q: -> -> int = &p;
10
11         printf("a = %d\n", a);
12         printf("b = %d\n", b);
13         printf("*p = %d\n", *p);
14         printf("**q = %d\n\n", **q);
15
16         // here some magic will happen ;-)
17
18         *p = 13;
19         printf("a = %d\n", a);
20         printf("b = %d\n\n", b);
21
22         **q = 31;
23         printf("a = %d\n", a);
24         printf("b = %d\n", b);
25 }
```

Aufgaben:

1. Beschreibe die Speicherbereiche (Memory-Locations) der Variablen **a**, **b**, **p**, **q**, wenn die folgenden Zeilen in der Funktion **main** erreicht werden: Zeile 10, 21, und 25.
2. Ersetze den Kommentar in Zeile 16 durch eine Anweisung, sodass in den Zeilen 18 und 22 jeweils die Variable **b** in der Zuweisung überschrieben wird.
3. Füge nach jeder Zeile **printf**-Anweisungen mit dem richtigen Platzhalter ein, um:
   A. die Größe der Variablen a, b, p und q auszugeben.
   B. die Adresse von "a" und "p" auszugeben.
   C. die Werte der Variablen p und q auszugeben.
4. Bestätige mit diesen Ausgaben von Teilaufgabe 3 deine Beschreibung des Speichers in Teilaufgabe 1, wenn die Zeile 10 in der Funktion "main" erreicht wird.

Schritt 2: Type-Cast
Hier ist das Programm **ex2.abc** mit einigen Kommentaren und Anweisungen:

```
 1 @ <stdio.hdr>
 2
 3 fn main()
 4 {
 5     local a: int = 0x12345678;
 6
 7     printf("a = 0x%x\n", a);
 8
 9     local ch: -> char = "ABCD";
10
11     printf("ch[0] = 0x%hhx\n", ch[0]);
12     printf("ch[1] = 0x%hhx\n", ch[1]);
13     printf("ch[2] = 0x%hhx\n", ch[2]);
14     printf("ch[3] = 0x%hhx\n", ch[3]);
15 }
```

Bemerkung:
Hier werden die Platzhalter "%x" und "%hhx" für den Formatstring verwendet:
- Der Platzhalter "%x" wird verwendet, um einen int-Wert in Hexadezimaldarstellung auszugeben.
- Der Platzhalter "%hhx" wird verwendet, um einen char-Wert in Hexadezimaldarstellung auszugeben.

Aufgaben:
1. Beschreibe die Speicherbereiche (Memory-Locations) von **a** und **ch**, **ch[0]**, **ch[1]**, **ch[2]**, **ch[3]**, wenn die Zeile 10 in der Funktion erreicht wird.
2. Ändere die Zeilen 11 bis 14 so, dass die Zeichen ']' und '[' nicht mehr vorkommen.
3. Ändere Zeile 9 zu:

   ```
   9     local ch: -> char = &a;
   ```

   - Der Compiler wird jetzt zum Complainer. Erkläre, warum die Fehlermeldung begründet ist.
   - Nehmen wir an, dass der Compiler das Programm doch übersetzt hätte. Beschreibe, wie die Speicherbeschreibung dann nach Zeile 9 aussehen würde.
4. Der Compiler führt für uns eine Art von Buchhaltung und merkt sich für jede Variable deren Typ. So weiß der Compiler, dass der Typ von **&a** ein Zeiger auf eine Variable vom Typ **int** ist. Mit einem (expliziten) Type-Cast können wir in diese Buchhaltung selbst eingreifen. Das geschieht, wenn wir Zeile 9 wie folgt ändern:

   ```
   9     local ch: -> char = (-> char)&a;
   ```

   In den runden Klammern wird (explizit) ein Datentyp angegeben, der für den Ausdruck rechts davon angenommen werden soll:
   A. Übersetze das Programm und führe es aus.
   B. Beschreibe wie in der Speicherfläche (Memory-Location) von a die einzelnen Bytes abgespeichert sind.
5. Mit dem Programm **objdump** kann der Maschinencode eines ausführbaren Programms ausgegeben werden. Für die nächste Aufgabe ist es hilfreich, wenn ihr dieses Programm jetzt bereits für die ausführbare Datei in dieser Aufgabe testet.

   Die Ausgabe von **objdump** zeigt den Maschinencode des ausführbaren Programms in einer hexadezimalen Darstellung sowie den entsprechenden Assembler-Code. Diese Ausgabe kann je nach Plattform und Compiler variieren, aber das Grundprinzip bleibt dasselbe.

   Wurde **ex2.abc** zum Beispiel in **a.out** übersetzt, liefert der Befehl **objdump -d a.out** eine Ausgabe ähnlich der folgenden:

   ```
   …
   0000000100000ed0 <_main>:
   100000ed0: 48 83 ec 18                 subq    $24, %rsp
   100000ed4: c7 44 24 0c 78 56 34 12     movl    $305419896, 12(%rsp)    ## imm = 0x12345678
   …
   ```

   Die genaue Form der Ausgabe kann variieren, aber in der Regel besteht sie aus einer hexadezimalen Adresse gefolgt von den Maschinenbefehlen und deren entsprechender Darstellung in Assembler-Code.

Schritt 3: Adressen und Maschinenbefehle
Vom Compiler wird ein Programm in eine Folge von Maschinenbefehlen übersetzt. Bei der Ausführung werden diese sequenziell ausgeführt. Die Befehle einer Funktion liegen bei der Ausführung in einem Speicherblock. Wird der Adressoperator auf einen Funktionsnamen angewandt, dann liefert dieser Ausdruck die Adresse der ersten Instruktion in dieser Funktion.

Verwendet folgendes Programm **ex3.abc**, um die Adresse von main und die ersten 20 Bytes der Funktion main auszugeben:

```
@ <stdio.hdr>

fn main()
{
    local ch: -> char = (-> char)&main;

    printf("ch = %p\n", ch);
    for (local i: int = 0; i < 20; ++i) {
        printf("%hhx ", ch[i]);
    }
    printf("\n");
}
```

Vergleicht die Ausgabe mit der von **objdump**, um die Übereinstimmung der Adresse und der ersten Bytes von **main** zu überprüfen.

Schritt 4: Funktionszeiger
Da jede Funktion eine Adresse hat, kannst du diese in einer (Pointer-)Variable speichern. Der Pointertyp sollte dann für den Elementtypen (für das, was am Ende des Zeigers liegt) die sogenannte Signatur der Funktion haben. Die Signatur einer Funktion beschreibt die Anzahl und Typen der Parameter sowie den Typ des Rückgabewertes.

Schau dir dazu das Beispiel **ex4.abc** an:

```
 1 @ <stdio.hdr>
 2
 3 fn german(wert: int): int
 4 {
 5     printf("Hallo Welt! wert = %d\n", wert);
 6     return wert + 42;
 7 }
 8
 9 fn english(value: int): int
10 {
11     printf("hello, world! value = %d\n", value);
12     return value + 123;
13 }
14
15 fn main()
16 {
17     local msg: -> fn(val: int): int;
18
19     msg = &german;
20
21     local foo: int = (*msg)(12);
22
23     printf("foo = %d\n", foo);
24 }
```

1. Übersetzt das Programm und testet es.
2. Ändere das Programm so, dass nach Zeile **msg** auf die Funktion **english** zeigt und somit in Zeile 21 diese Funktion aufgerufen wird.
3. Für die Signatur spielt der verwendete Bezeichner einer Variabel keine Rolle. Deshalb ist in Zeile 17 der Bezeichner **val** nicht relevant (macht den Code aber eventuell lesbarer). Bestätige, dass Zeile 17 geändert werden kann in

   ```
   local msg: -> fn(: int): int;
   ```

4. Funktionen können natürlich auch Zeiger zurückgeben, und dies können natürlich Zeiger auf Funktionen sein. Verwende folgende Funktion selectLang in deinem Programm, um in Zeile 19 die Variable **msg** zum Beispiel mit der Adresse von **german** zu initialisieren:

```
fn selectLang(sel: int): -> fn(: int): int
{
    if (sel == 0) {
        return &german;
    } else {
        return &english;
    }
}
```

5. Mit der Funktion **scanf** (deklariert in **stdio.hdr**) kann ein Variable eingelesen werden, wie in diesem Beispiel **test_scanf.abc**:

```
@ <stdio.hdr>

fn main()
{
    local a: int;

    printf("a = ");
    scanf("%d", &a);
    printf("a = %d\n", a);
}
```

   Übersetzt und testet diese Beispiel. Erkläre wieso der Funktion **scanf** die Adresse von **a** übergeben werden muss.

6. Ändere das Programm so ab, dass du zunächst zwei Integer-Werte eingeben kannst:
   - Mit dem ersten Wert soll ausgewählt werden, ob **german** oder **english** aufgerufen wird.
   - Mit dem zweiten Wert soll das Argument für den Funktionsaufruf festgelegt werden.

   Dein Programm sollte dann zum Beispiel so verwendet werden können und eine entsprechende Ausgabe erzeugen:

```
MCL:tmp lehn$ ./a.out
lang = 1
val = 34
hello, world! value = 34
result = 157
MCL:tmp lehn$ ./a.out
lang = 0
val = 123
Hallo Welt! wert = 123
result = 165
```