# Programmering for computerteknologi
# Hand-in Assignment Exercises

## *Week 4: Array data structures*

Please make sure to submit your solutions **by next Monday**.

In the beginning of each question, it is described what kind of answer that you are expected to submit. If *Text and code answer* is stated, then you need to submit BOTH some argumentation/description and some code; if just (*Text answer*) or (*Code answer*) then just some argumentation/description OR code. The final answer to the answers requiring text should be **one pdf document** with one answer for each text question (or text and code question). Make sure that you have committed your code solutions to that repository and submitted to Brightspace as well.

*Note:* the **Challenge** exercises are *optional*, the others mandatory (i.e. you **have** to hand them in).

### Exercises

(1) (Text and code) When asking ChatGPT the following question

> write a program that can print out a repayment plan for student loans

I got the following answer

```c
#include <stdio.h>
#include <math.h>

int main() {
    double loanAmount;   // Loan principal amount
    double interestRate; // Annual interest rate (as a decimal)
    int loanDuration;    // Loan duration in months

    // Get user input
    printf("Enter the loan amount: ");
    scanf("%lf", &loanAmount);

    printf("Enter the annual interest rate (as a decimal): ");
    scanf("%lf", &interestRate);

    printf("Enter the loan duration in months: ");
    scanf("%d", &loanDuration);

    // Calculate monthly interest rate
    double monthlyInterestRate = interestRate / 12;

    // Calculate monthly payment
    double monthlyPayment = loanAmount * (monthlyInterestRate / (1 - pow
    (1 + monthlyInterestRate, -loanDuration)));

    // Display repayment plan
```

```
26      printf("\nRepayment Plan:\n");
27      printf("Month\tPrincipal\tInterest\tTotal Payment\n");
28
29      double remainingBalance = loanAmount;
30      int month = 1;
31
32      while (1) {
33          double interestPayment = remainingBalance * monthlyInterestRate;
34          double principalPayment = monthlyPayment - interestPayment;
35
36          remainingBalance -= principalPayment;
37
38          printf("%d\t%.2lf\t\t%.2lf\t\t%.2lf\n", month, principalPayment,
    interestPayment, monthlyPayment);
39
40          if (++month > loanDuration || remainingBalance <= 0.0) {
41              // Break the loop when the loan is paid off or the loan term
    is exceeded
42              break;
43          }
44      }
45      return 0;
46  }
```

(a) Make at least four test cases for the program

(b) Use the test cases to ensure the the program works as expected.

(c) Add pre- and post conditions to the code and include them in your code as `assert`-statements

(d) Change the `while`-loop to a `for`-loop

(e) Refactor the program so that it has two functions: `calculateMonthlyPayment` (returns the monthly payment) and `displayRepaymentPlan` that prints out the repayment plan). Remember to make and include pre- and post conditions for the functions

(2) Complete the function that returns the smallest element in an array:

```
1  /*
2   * Returns the smallest of the first n values in list
3   * Pre: n>0, list[0...n-1] is defined
4   */
5  int get_min( int list[], int n) {
6    assert(n>0);
7    ...
8  }
```

*Hint:* see listing 7.6 in the book.

(3) Complete the following function that returns an arry in reverse order:

```
1  /*
2   * Returns in rev_array the elements of list in reversed order
3   * Pre: n>0, list[0...n-1] is defined
```

```
 */
void reverse( int list[], int rev_array[], int n) {
  assert(n>0);
  ...
}
```

For example, given the arrays:
`int a[5] = {1,2,3,4,5};`
`int b[5];`
and calling the function in the following way:
`reverse(a, b, 5);`
...you should end up with the array `b` that has the elements: `[5,4,3,2,1]`.
**Note**: A function cannot return an array, that is why the second array (`rev_array`) is used as the output parameter (and `list` as the input parameter)

(4) Suppose I have a sequence of numbers such as:
$1, 7, 43, 4, 67, 0, 3, 2, 0, 0, 3, 2, \mathbf{0,0,0}, 3, 2, 6$
The longest sequence of zeroes has 3 consecutive zeroes (in bold). Write a function that computes the start index of the longest sequence of zeros in an array. The function should have the following signature:

```
/*
 * Returns the index in list of the logest sequence of zeros in list, -1
    if no zeros in list
 * pre: n>0, list[0...n-1] is defined
 */
int longest_seq(int list[], int n){
  assert(n>0);
  ...
}
```

Given the following lines of code:

```
  int a[13] = { 0, 0, 0, 4, 5, 0, 0, 0, 0, 0, 11, 0, 0 };
  int b[5] = {1, 2, 3, 4, 5};
  printf ("The longest sequence of zeros start index is %d\n",
    longest_seq(a,13));
  printf ("The longest sequence of zeros start index is %d\n",
    longest_seq(b,5));
```

the program shold print the following:
`The longest sequence of zeros start index is 5`
`The longest sequence of zeros start index is -1`

(5) Write a function that counts the occurrences of numbers between 1 to 20 in a two-dimensional array of size $100 \times 150$, with the following signature:

```
/*
* pre:  a contains numbers between 1..20
* post: count[i] is equal to the numers af i+1 in a
*/
void count_1_to_20(int a[100][150], int count[20]) {
   ...
   return ;
```

```
}
```

**Note** `count[i]` contains the number of `i+1` in `a`

(6) Complete the following function that returns the average of an array of integers:

```
/*
 * Returns the average of an array
 * Pre: n>0, list[0...n-1] is defined
 */
double average( int list[], int n) {
  assert(n>0);
  ...
}
```

(7) **Challenge:** Write 1-2 sentences that answers the question: What does the following program compute, that is, what problem does it solve?

```
int r = 0;
int n = 0;
int s = 0;
int h;
while (n != N) {
  h = n;
  while (h != s) {
    if (a[h - 1] != a[n])
      h--;
   else
      s = h;
  }
  r = max(r, n + 1 - s);
  n++;
 }
 printf("%d", r);
}
```

where the function `max()` is given by

```
int max(int x, int y) {
  return (x >= y) ? x : y;
}
```

*Hint:* Experiment with different values for $a$ and $N$, e.g.

```
int a[11] = { 9,1,1,7,3,4,2,11,1,9,10 };
int N = 11;
```

(8) **Challenge:** The **Game of Life** is a cellular automaton (it looks a bit like a board game) that demonstrates how very simple rules applied on a large scale can create complex *emergent* patterns. It was invented by a mathematician John Conway. The idea is that the universe in the game of life is a checkerboard (we can suppose it is a fixed size of $n \times n$), we call each square on the board a *cell*. Each cell is either alive (represented by $*$) or dead (represented by $-$). Cells interact with their *neighbours*, which are the eight surrounding cells (above, below, left, right, upper-left, etc.). On each turn (called a *generation*), whether a cell is alive or not can change according to the following rules:

**Rule 1:** An alive cell with less than two alive neighbours dies (under population)

**Rule 2:** An alive cell with exactly two or three alive neighbours stays alive, and lives on into the next generation (survival)

**Rule 3:** An alive cell with more than three alive neighbours dies (over population)

**Rule 4:** A dead cell with exactly three alive neighbours becomes a live cell (reproduction)

The rules are illustrated in the following examples by the cell in the middle of each board.

**Rule 1: under population**

```
Generation 1:
- - -
- * -
- - -

Generation 2:
- - -
- - -
- - -
```

**Rule 2: Survival**

```
Generation 1:
- * -
- * *
- - -

Generation 2:
- - -
- * -
- - -
```

**Rule 3: Over population**

```
Generation 1:
- * -
* * *
- * -

Generation 2:
- - -
- - -
- - -
```

**Rule 4: Reproduction**

```
Generation 1:
- * -
- - *
```

```
* - -

Generation 2:
- - -
- * -
- - -
```

The game is played by setting an initial configuration of alive and dead cells on the board and watching how the cells evolve over time. When certain configurations of alive and dead cells appear on the board, some interesting patterns start to emerge. For example, the following pattern (called a bee-hive) is stable, and doesn't change between generations (a category of patterns called *Still Life*):

**Example: Bee-hive**

```
Generation 1:
- - - - - -
- - * * - -
- * - - * -
- - * * - -
- - - - - -

Generation 2:
- - - - - -
- - * * - -
- * - - * -
- - * * - -
- - - - - -
```

The following pattern (called a *blinker*) cycles through its states in a repeating fashion (a category of patterns called *Oscillators*):

**Example: Blinked**

```
Generation 1:
- - - - -
- - * - -
- - * - -
- - * - -
- - - - -

Generation 2:
- - - - -
- - - - -
- * * * -
- - - - -
- - - - -

Generation 3:
- - - - -
- - * - -
- - * - -
- - * - -
- - - - -
```

And finally, there are a category of patterns that actually move across the board called *Spaceships*, the most famous being the *glider*. The Wikipedia page has some nice visualisations: `https://en.wikipedia.org/wiki/Conway%27s_Game_of_Life`

Your task is to program the game of life in C. Create an $n \times n$ board as a char array for an appropriate $n$ (large enough to see some nice patterns, small enough to display well on the terminal). Have your program print out the board of each generation, and let it run for a few generations (perhaps 10). Write a test for your program that demonstrates:

**(a)** a still life pattern
**(b)** an oscillator pattern
**(c)** a spaceship