Exercise 2.1 — Treasure Hunt (45% of Midterm 1, 2003)

Design and implement a C program to play the “blind” board game “treasure hunt”.

- The board has 20 × 20 fields, from (1, 1) to (20, 20).
- On field (17, 2) there is a treasure.
- The player starts on field (9, 10), but is not told this.
- All fields \((x, y)\) with \((x + 2y)\) divisible by 5 are forbidden, i.e., the player must not be allowed to move onto such a field.
- The player navigates the board by entering “numeric keypad cursor control commands”:
  - “2” moves down one step
  - “8” moves up one step
  - “4” moves left one step
  - “6” moves right one step

After each successful move, only the new distance to the treasure is displayed — for this, the 1-norm is used and whether a field is forbidden or not does not matter, so, e.g., the distance from (9, 10) to (17, 3) is 15 (calculated as 8 + 7).

- When the player tries to move off the board or onto a forbidden field, a message is displayed noting that the move is impossible, but not why it is impossible.
- When the player moves to the field where the treasure is, a congratulatory message is displayed and the program terminates.

Assume that the user will input only numbers! Do not use arrays!

Solution Hints

Design:

- State: integer coordinates.
- Structure: loop until treasure found:
– Input direction
– Calculate hypothetical new position into auxiliary variables
– Check whether new position is legal:
  If yes, move there and output new distance;
  if no, output error message that does not give too much away.

Implementation:

```c
#include <stdio.h>
int main()
{
    int target_x=17, target_y=3;

    int x_max=20, y_max=20;
    int x=9, y=10;
    int input, new_x, new_y;
    char *message; /* superfluous luxury */

    while ( x ≠ target_x || y ≠ target_y )
    {
        scanf("%d", &input);
        new_x = x; new_y = y;

        switch(input) {
            case 4: new_x = x-1;
                        message = "cannot move left";
                        break;
            case 6: new_x = x+1;
                        message = "cannot move right";
                        break;
            case 2: new_y = y-1;
                        message = "cannot move down";
                        break;
            case 8: new_y = y+1;
                        message = "cannot move up";
                        break;
            default: printf("???
");
        }

        if (new_x > 0 && new_x ≤ x_max &&
                new_y > 0 && new_y ≤ y_max && ((new_x + 2 * new_y) % 5 ≠ 0))
        {
            x = new_x;
            y = new_y;
            printf("Your distance to the treasure: %d\n",
                   abs(target_x - x) + abs(target_y - y));
        }
        else
        {
```
```c
    printf("\%s\n", message);
}
}
    printf("Congratulations! You found the treasure at (%d,%d).\n", x, y);
    return 0;
}

Exercise 2.2 (Textbook Exercise Recommendation)
Read chapter 4 of the textbook. Do at least the following exercises: 4.5–4.14, 4.24, 4.29

Solution Hints
The last two are about Boolean operations and De Morgan — check the “C-Truth” slides and your logics material if you have any problems.

Exercise 2.3 — ASCII Art: Zig-Zag — (50% of Midterm 1, 2004)
Design and implement a C program that asks the user for a height, and for two offset numbers, and uses these three numbers to print a combination of two zig-zag lines of the same height, as in the following example:

Note that one of the zig-zag lines is drawn using the “plus” symbol, the other using the letter “X”, and where both zig-zag lines intersect, the asterisk “*” is used.

The grid lines are of course not part of the output. Here is another example without those grid lines — any such pattern should be producable:

Assume that the user will input only numbers! Do not use arrays!
Decompose into functions! Design and Document!

Solution Hints
“Only numbers” includes non-positive (or at least negative) numbers, which at least for height does not make sense — this has to be caught. If the offset numbers can be negative, this has to be documented.
My Design:

• Decisions:
  – Width is constant 79
  – “Offset” means how far from the left margin is the first entry in the first row. Taken strictly,
    this implies that offsets lie in the interval $[0, 2 \times height - 3]$. For input, I restrict the offsets to
    non-negative numbers, although the modulo calculation would be unaffected by that.

• Solution Structure:
  – Input three numbers $height$, $offset_1$, $offset_2$:
    — function $ask$ takes as argument the minimal acceptable number and insists on input until the
      entered number acceptable; that number is then returned.
  – Loop $height$ times for the rows, and $width$ times for the columns; each time:
    – deciding for each of the two zigzag lines whether they cross the current position (function
      $onZigZag$), and
    – printing the corresponding character
  – Before the whole loop, and after each row, print a new-line character.

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

int ask(int); /* interactively obtains from user a number bounded from below */

bool onZigZag (int height, int offset, int x, int y);
/* returns true if (x,y) is on the zigzag defined by height and offset */

int main() {
    int height, offset1, offset2;
    const int width = 79;
    int x,y;
    bool hit1, hit2;

    printf ("For the height of the zig-zag,\n");  height = ask(1);
    printf ("For the offset of the first zig-zag,\n");  offset1 = ask(0);
    printf ("For the offset of the second zig-zag,\n");  offset2 = ask(0);

    printf("\n");
    for ( y = 0; y < height; y++) {
        for ( x = 0; x < width; x++) {
            hit1 = onZigZag( height, offset1, x, y );
            hit2 = onZigZag( height, offset2, x, y );
            if ( hit1 ) {
                if ( hit2 ) printf("*"); /* hit1 && hit2 */
                else      printf("+"); /* hit1 */
```
else {
    if (hit2) printf("X"); /* hit2 */
    else printf(" ");
}
} /* end for(x) */
printf("n");
} /* end for(y) */
return 0;
}

int ask(int min) {
    int n = 0;
    do {
        printf("enter a number greater or equal to %d: ", min);
        scanf("%d", &n);
    } while (n < min); /* input that is too small leads to re-prompt */
    return n;
}

bool onZigZag(int height, int offset, int x, int y) {
    int period = 2 * (height - 1); /* length of zig-zag period */
    int local = (x + period - offset) % period; /* position in current period */
    if (local < height) /* falling flank */
        return local == y;
    else /* rising flank without ends */
        return (period - local) == y;
}

Exercise 2.4
What is the output of the following C program (which prints not more than ten lines):

#include <stdio.h>

int main ( void ) {
    char input[] = "terasse";
    char result[] = " "; // six spaces
    int i, j = 0, c = 3, q;
    for ( q = 3; q ≥ 0; q = q - c ) {
        for ( i = 0; i < c; i++ ) {
            printf("j = %d	c = %d	q = %d	i = %d\n", j, c, q, i);
            result[j] = input[q + i];
            j = j + 1;
        }
        c = c - 1;
    }
}
```c
printf("%s
", result);
return 0;
}
```

What is the value of `q` after termination of the outer loop?

**Solution Hints**

<table>
<thead>
<tr>
<th><code>j</code></th>
<th><code>c</code></th>
<th><code>q</code></th>
<th><code>i</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>3</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

`assert!`

The program terminates with the following two states before and after the closing brace of the outer loop:

<table>
<thead>
<tr>
<th><code>j</code></th>
<th><code>c</code></th>
<th><code>q</code></th>
<th><code>i</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>-65536</td>
<td>2147450880</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>-65536</td>
<td>-2147450880</td>
<td>0</td>
</tr>
</tbody>
</table>

**Note:**

\[
\begin{align*}
2^{16} & = 65536 \\
2^{31} & = 2147483648 \\
2^{31} - 2^{15} & = 2147450880
\end{align*}
\]

This program only terminates because of int wrap-around!