Exercises taken from E. Robers, *Programming Abstractions in C++* (2013).
Use the C++14 standard in solving your problems.
Always compile with `-Wall -Wextra -Werror` to detect warnings and errors.

Basics:

1. Write a program that reads in a temperature in degrees Celsius (C) and displays the corresponding temperature in degrees Fahrenheit (F). The conversion formula is
$$F = \frac{9}{5}C + 32$$

2. Write a program that reads in a positive integer N and then calculates and displays the sum of the first N odd integers. For example, if N is 4, your program should display the value 16, which is $1 + 3 + 5 + 7$.

3. The German mathematician Leibniz discovered the rather remarkable fact that the mathematical constant π can be computed using the following mathematical relationship:
$$\frac{\pi}{4} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \frac{1}{11} + \cdots$$
The formula to the right of the equal sign represents an infinite series; each fraction represents a term in that series. Write a program that calculates an approximation of π consisting of the first 10000 terms in Leibniz's series.

Functions:

1. If you did not do so the first time around, rewrite the Celsius-to-Fahrenheit program from exercise 1 so that it uses a function to perform the conversion.

2. Write a function `round_to_nearest_int(x)` that rounds the floating-point number x to the nearest integer. Show that your function works by writing a suitable main program to test it.

3. Write a program `random_ave()` that repeatedly generates a random real number between 0 and 1 and then displays the average after a specified number of trials entered by the user.

Pointer and arrays:

1. What is the purpose of the sizeof operator? How do you use it? [Check cplusplus.com]

2. What are the types of the variables introduced by the following declaration: `int * p1, p2;`

3. Assuming that variables of type int and all pointers require four bytes of memory, draw a diagram showing a portion of the stack frame that contains the following declarations:

```
int v1 = 10;
int v2 = 25;
int *p1 = &v1;
int *p2 = &v2;
```

In your diagram, trace through the operation of these statements:

```
*p1 += *p2;
p2 = p1;
*p2 = *p1 + *p2;
```

4. Write array declarations for the following array variables:

   a) An array `real_array` consisting of 100 floating-point values

   b) An array `in_use` consisting of 16 Boolean values

   c) An array `lines` that can hold up to 1000 strings

   Remember to declare constants to specify the allocated size for these arrays.

5. If array is declared to be an array, describe the distinction between the expressions `array[2]` and `array + 2`.

6. Describe the effect of the idiomatic C++ expression `*p++`.

7. Write an implementation of the selection sort algorithm so that it sorts an array rather than a vector. Your function should use the prototype `void sort(int array [], int n)`.

8. Using memory diagrams, explain the difference between the following instructions and discuss if any error is generated.

```
int ival = 1024;
int &ref;
int &rval = ival;
int *pval1 = rval;
int *pval2 = &rval;
int *pval3 = ival;
int *pval4 = &ival;
```

9. Explain what types of data the following statements generate.

```
bool array[128];
bool *ap[128];
```

Dynamic memory:

1. What is a memory leak? [Check Wikipedia.]

2. Write a function `create_index_array(n)` that allocates a dynamic array of n integers in which each integer is initialized to its own index.

3. Write a function `char *copy_C_str(char *str)` that allocates enough memory for the C-style string str and then copies the characters—along with the terminating null character—into the newly allocated memory.

4. Write a function to concatenate two literals types `string`, assign the concatenation and return the result in a dynamically assigned `char` array. The prototype of the implemented function must be

```
... concatenate(const string &str1, const string &str2);
```

   where the points `...` indicate a design decision that must be made regarding the data type that must be returned. Make sure that the dynamically reserved memory is properly removed from the stack (or is it heap?). Write a couple of test cases and show that the function performs the task and dynamic memory is handled correctly.

5. Write a function that returns a dynamically allocated vector of `int`s. Pass that vector to another function that reads the standard input to give values to the elements. Pass the vector to another function to print the values that were read. Remember to delete the vector at the appropriate time.

6. Explain what if anything is wrong with the following function.

```
bool b() {
    int* p = new int;
    // ...
    return p;
}
```

7. Write a program that reads a `string` from the standard input into a dynamically allocated `char`acter array. Describe how your program handles varying size inputs. Test your program by giving it a `string` of data that is longer than the array size you've allocated.

8. Rewrite all previous exercises now using C's `malloc`, `calloc`, `realloc`, and `free`.

9. Using C++'s `new` and `delete`, implement your own versions of C's `malloc`, `calloc`, `realloc`, and `free`.