NSWI170 – **Computer Systems** – 2024/25 Summer – Martin Svoboda

Assignment **A1: Celmomether**

101. [**Code decomposition**] In general, it is necessary to appropriately decompose any sufficiently non-trivial problem into smaller and simpler units, each of which should solve one partial and logically well-defined and bounded task. It is one of the key aspects of writing good code, we will, therefore, strictly abide by it from the very beginning.

102. [**Decomposition using functions**] Within this particular assignment, we will suffice with the decomposition based only on the design of individual functions. If we were to solve more complex tasks, we would proceed similarly for larger units at the level of entire files, modules, etc.

103. [**Code universality**] When designing our functions, we should always strive to achieve a slightly higher level of generality than it would be absolutely necessary. At least, we should be able to call our functions repeatedly, from different places, as well as with different values of their input parameters.

104. [**Excessive decomposition**] Excessively fine-grained decomposition would also be a mistake, though. It means creating functions that would be so simple or logically incomplete that their separation would not actually bring any benefits. We should also realize that the execution of a function call is actually a relatively time-consuming operation.

105. [**Inappropriate optimization**] In other words, we should generally not forget about the aspect of code optimization in terms of its speed. However, achieving this goal should never come at the expense of quality and good decomposition, as these are more important from the point of long-term code maintainability.

106. [**Order of function parameters**] When it comes to the interface of our functions, we should also pay attention to finding a suitable order of their individual intended parameters. Although it may not always be technically possible to do so, the most reasonable approach usually is to proceed according to their logical importance.

107. [**Good design indicator**] Initial indicator of a good design often is the ability to find appropriate names, in our case, names of functions and also their parameters. On the contrary, weird, unspecific or excessively long names should be a warning.

108. [**Quality names**] This brings us to the general principle that names of all functions, their parameters, local and global variables, constants, and, in fact, anything else should be reasonably brief, yet concise, specific, and self-describing in terms of their meaning and expected use.

109. [**Abbreviations in names**] We should also make sure we have prevented unwanted alternative interpretations of our names. Except in justified cases, we should therefore not use abbreviations at all. For example, `temp` could mean *temperature* but also *temporary*, or, analogously, `val` can be understood as *value* or *valid*. On the contrary, iterative variables such as `i` in loops are all right.

110. [**Naming convention**] It is not that important what particular naming convention we use, but we must always be consistent throughout our code. However, functions, variables, and parameters are usually written with lowercase initial letters. We also need to think about the way how to write multi-word names.

111. [**Usage of English**] Since we nowadays often work in teams, especially international ones, when developing our applications, it is more than appropriate to write all the code automatically in English. And it is not just about the already discussed names, but also comments.

112. [**Source file structure**] At the beginning of our source file, we first provide the include directives for all the required libraries and header files. We then continue with global constants, if any. Finally, we define all the individual functions in such an order that we always have all the functions we want to actively use already defined earlier. Therefore, the `main` function will be at the very end.

113. [**Content of the main function**] Function `main` as such should generally not contain any complex or too low-level or technical code from the point of view of the application logic. In our case, we can basically limit ourselves just to calling a single function that will represent and encapsulate all the functionality of our unreliable thermometer.

114. [**Global variables**] We will do without any global variables in this particular assignment, we do not need them. In other words, we will design the interface of all functions and their return values in such a way that we have full control over the flow of data and all their changes.

115. [**Alternative inputs**] In order to technically simplify our situation, we assume the input temperatures are provided in the form of a global constant array. It is obvious, however, that we could obtain them in other ways as well, for example, via the arguments passed when starting the program, from the standard input, or even input files on a hard drive. We will, therefore, implement the main function of our thermometer so that we can possibly call it with different inputs, even repeatedly.

116. [**Hardwired values**] In other words, it is not possible to hardwire our input array of temperatures in our executive code in any way. That applies to the `no_value` constant for invalid temperatures, too. Therefore, the only place we will refer to these original inputs will be in the `main` function.

117. [**Input array modifications**] We will work with the provided array of temperatures in a read-only mode. Moreover, will not solve our task by making a copy of this input array first and somehow modifying or otherwise preprocessing the individual values within it. In other words, we do not want to change the content and meaning of the input data, we just want to use and process it as expected.

118. [**Types of constants**] As for constants in general, let us add that constructs `const` and `constexpr` are technically different. While the first case is basically just an ordinary variable with value that cannot be modified, the latter case is a value that is constant even at compile time. We will thus use them whenever possible because they can lead to more efficient code.

119. [**Array size**] The traditional C-style array we use for our temperatures does not know its size by itself. This means we always need to pass this information appropriately together with the array as such.

120. [**Temperature array size**] At the same time, both of these pieces of data logically belong to each other by their nature. Hence, we should treat them symmetrically even from the technical point of view, that is, in terms of their scope of definition (global vs. local) as well as their constancy form.

121. [**Passing array parameter**] Although a C-style array is actually just a pointer to its first element, it is still better to declare it as, for example, `int values[]` and not `int * values` in a function parameter. At first glance, it will therefore be clear that we really intend to work with an array and not just one single value.

122. [**Array parameter size**] Although it is seemingly possible to describe the size of an array within the description of the expected function parameters and so write, e.g., `int values[size]`, this size will in fact be entirely ignored and, therefore, cannot be used for this purpose.

123. [**Checking array bounds**] Whenever we work with an array, it is our own responsibility to ensure that we access its existing elements only. In other words, attempts to access invalid positions before its beginning or beyond its end are not checked and handled for us. In the best case, such an error leads to an immediate crash of the entire program (signal 11 *segmentation violation*).

124. [**Number of passes**] We should always try to process the input data using only the absolutely necessary number of passes, ideally just one. This will obviously not be possible in our case due to the temperature graph alignment, but we avoid unnecessary passes even so.

125. [**Repeated calculations**] Similarly, we avoid unnecessarily repeated partial calculations that only need to be performed once because they are deterministic, i.e., they always return the same result over the same inputs.

126. [**Minimal temperature value**] When looking for the minimal temperature to align the temperature graph, it is advisable to think carefully about what default (or, in other words, the worst possible) value is assumed by the assignment as such. In any case, it is not possible to just make up some constants that are not assumed in the assignment, no matter how appropriate they could seem to us, especially large or small.

127. [**Invalid temperature value**] Invalid temperature is specified using a named constant `no_value`. As for the actual value of this constant, it can be arbitrary. So, for example, `-999`, but also any other, whether negative or positive. Again, we cannot work with assumptions whose validity has not been explicitly guaranteed to us.

128. [**Uniform processing of temperatures**] It is also reasonable to approach each element in the temperature array in the same way, none of the valid temperatures has any special status. Specifically, for example, just because it would be provided as the first element. Therefore, processing of all values should be equivalent from the code point of view.

129. [**Minimal vs. smallest temperature**] Finally, note that the minimal value is mathematically different from the smallest (lowest) value, so we should keep this in mind when naming.

130. [**Temperature graph alignment**] Interface of each function should be designed in a way that we expect only the relevant input parameters to be specified when calling it. In particular, we should not force the callers to resolve and give us any information that is actually an integral part of the task we are solving and so we should calculate it by ourselves.

131. [**Printing symbols**] It is obvious that we will often need to print various numbers of certain symbols when outputting the temperature graph. It is therefore apparent that we should prepare some auxiliary, sufficiently universal function for this purpose and then just use it repeatedly.

132. [**Repetition of similar code**] The previous idea is basically about nothing more than that we should always try to avoid unnecessary repetition of similar or even the same fragments of code. At least unless there would be special reasons for doing so, which is not the case for us now.

133. [**Excessive code branching and nesting**] When printing one particular temperature, it is advisable to avoid excessive overuse of conditional expressions, let alone complicated or nested ones. Especially since it is not needed in our situation. Instead of branching the main code, it is thus better to try to calculate the expected numbers of symbols using various operations or even the ternary operator.

134. [**Auxiliary round parentheses**] Although not always necessary, it is generally better to wrap individual conditions in more complicated logical expressions with round parentheses. Not only will this increase the overall code clarity, but, at the same time, we will not need to think about operator priorities, which may sometimes not be straightforward.

135. [**Calling a function with different parameters**] If we want to call a function, but depending on the situation with different parameter values, it is better to branch the code to determine these parameters rather than the main code, and thus call this function more than once. Again, the ternary operator could also help us here.

136. [**Last valid temperature**] When printing the graph of temperatures, it will be necessary to deal with situations when the current value is not valid. In such a case, it would not be a good idea to try to traverse the array in the opposite direction and somehow try to find the last previous valid temperature. Moreover, especially not using some recursive function.

137. [**Transferring temperatures across loop iterations**] If we need to preserve certain values through individual iterations of the loop, we limit ourselves to only the absolutely necessary information corresponding to the logical nature of the problem solved. We will, therefore, not transfer any calculated, technical or auxiliary data, but really only the temperature as such.

138. [**General quality principles**] More complicated parts of the code will be supplemented with brief comments. Adherence to all general principles of code quality is a matter of course. This means, for example, that we should not write the code for ourselves, but for someone else. That is, to create

such a code that it will be comprehensible to others. Including ourselves in a few years, when we will undoubtedly not remember anything.

139. [**Starter packages**] Starter packages for individual assignments available within ReCodEx are not intended for direct use or incorporation in our code. They only serve for basic orientation or explanation of some technical or other aspects.

140. [**Disallowed language resources**] We will avoid using any advanced constructs we have not learned as a part of this assignment. In particular, we will not use `std::string`, library `iostream`, etc. The reason is that we do not want to stray from our intention of low-level programming. Additionally, our hardware resources will be limited when working with Arduino in the following assignments, and so we will also always strive for adequately simple and efficient code.

141. [**Compilation warnings**] Although compilation warnings, unlike compilation errors, may not always indicate wrong code, their presence really signals at least potential problems or inappropriate code in most cases. We will, therefore, always resolve all such possible warnings for this reason, too.