

Úkol A1: Teploměr

101. [**Dekompozice kódu**] Každou dostatečně netriviální úlohu je obecně potřeba vhodně dekomponovat do menších a jednodušších celků, přičemž každý z nich by měl řešit jeden dílčí a dobře logicky vymezený a ohraničený úkol. Jde o jeden z klíčových aspektů psaní dobrého kódu, a proto jej budeme důsledně dodržovat už od samotného začátku.
102. [**Dekompozice pomocí funkcí**] V rámci tohoto konkrétního úkolu si při dekompozici vystačíme jen s návrhem jednotlivých funkcí. Pokud bychom řešili složitější úlohy, obdobně bychom postupovali i pro větší celky na úrovni celých souborů, modulů atp.
103. [**Univerzálnost kódu**] Při návrhu našich funkcí bychom se vždy měli snažit o dosažení nepatrně vyšší úrovně obecnosti, než je nezbytně nutně vyžadováno. Minimálně bychom měli být schopni naše funkce volat opakovaně, z různých míst nebo třeba s jinými hodnotami jejich vstupních parametrů.
104. [**Přehnaná dekompozice**] Chybou by ale také byla přehnaně jemná dekompozice, kdy bychom vytvořili natolik jednoduché nebo logicky neucelené funkce, že by jejich vyčlenění ve skutečnosti žádné výhody nepřinášelo. Měli bychom si také uvědomit, že samotná realizace volání jakékoli funkce je časově poměrně náročnou operací.
105. [**Nevhodná optimalizace**] Jinými slovy bychom v obecné rovině neměli zapomínat ani na aspekt optimalizace kódu z hlediska jeho rychlosti. Dosažení tohoto cíle by ale nikdy nemělo být na úkor kvality a dobré dekompozice, protože ty jsou z pohledu dlouhodobé udržitelnosti kódu důležitější.
106. [**Pořadí parametrů funkcí**] Pokud jde o rozhraní našich funkcí, pozornost bychom měli věnovat i nalezení vhodného pořadí jejich jednotlivých zamýšlených parametrů. Přestože takový přístup nemusí být vždy technicky možný, nejrozumnější obvykle bude vyjít z jejich logické důležitosti.
107. [**Indikátor dobrého návrhu**] Prvotním ukazatelem dobrého návrhu často bývá i schopnost nalezení vhodných názvů, v našem případě tedy názvů funkcí a vlastně i jejich parametrů. Naopak podivné, neurčité nebo až příliš dlouhé názvy by měly být varováním.
108. [**Kvalitní názvy**] To nás přivádí k obecnému principu, kdy názvy všech funkcí, jejich parametrů, lokálních i globálních proměnných, konstant a vlastně i čehokoli dalšího by měly být rozumně stručné, přesto výstižné, specifické a samovysvětlující z hlediska jejich významu a očekávaného použití.
109. [**Zkratky v názvech**] Také bychom měli mít jistotu, že jsme zabránili nechtěným alternativním interpretacím našich názvů. Až na odůvodněné případy bychom tedy vůbec neměli používat zkratky. Například `temp` může znamenat *temperature* ale také *temporary*, analogicky `val` můžeme chápat jako *value* nebo *valid*. Naopak iterační proměnné jako `i` apod. v cyklech jsou bezproblémové.
110. [**Pojmenovávací konvence**] Není až tak důležité, jakou konkrétní konvenci pojmenovávání budeme používat, vždy však musíme být v celém našem kódu konzistentní. Funkce, proměnné a parametry se nicméně obvykle píšou s malými počátečními písmeny. Také je potřeba promyslet způsob zápisu víceslovných názvů.
111. [**Použití angličtiny**] Jelikož v dnešní době často na vývoji aplikací pracujeme v týmech, navíc mezinárodních, je více než vhodné veškerý kód psát automaticky v angličtině. A nejde jenom o již diskutované názvy, ale také o komentáře.
112. [**Struktura zdrojového souboru**] Na začátku našeho zdrojového souboru nejprve uvedeme direktivy na načtení všech potřebných knihoven a hlavičkových souborů. Následně pokračujeme případnými globálními konstantami. Teprve pak definujeme jednotlivé funkce, a to v takovém pořadí, abychom vždy měli definované všechny ty funkce, které již aktivně chceme používat. Funkce `main` tedy bude na úplném konci.

113. [**Obsah funkce main**] Funkce `main` jako taková by obecně neměla obsahovat žádný složitý nebo z pohledu logiky aplikace příliš nízkourovňový nebo technický kód. V našem případě se v podstatě můžeme omezit jen na volání jedné jediné funkce, která zapouzdří veškerou funkcionalitu našeho nespolehlivého teploměru.
114. [**Globální proměnné**] V rámci tohoto úkolu se zcela obejdeme bez globálních proměnných, nebudeme je potřebovat. Jinými slovy rozhraní všech funkcí a případné návratové hodnoty navrhne tak, abychom nad tokem a změnami dat měli plnou kontrolu.
115. [**Alternativní vstupy**] Abychom měli technicky co nejjednodušší situaci, máme vstupní teploty zadané formou globálního konstantního pole. Je ale zřejmé, že bychom je mohli získat i jinak, třeba z argumentů předaných při spuštění programu, ze standardního vstupu nebo i nějakého vstupního souboru na disku. Hlavní funkci našeho teploměru tedy naprogramujeme tak, abychom ji případně mohli zavolat i s jinými vstupy, a to i opakovaně.
116. [**Zadrátované hodnoty**] Jinými slovy není možné naše vstupní pole teplot jakkoli ve výkonném kódu napevno zadrátovat. To platí i pro konstantu `no_value` pro neplatné teploty. Jediné místo, kde se na tyto naše původní vstupy odkážeme, tedy bude ve funkci `main`.
117. [**Úpravy vstupního pole**] S polem zadaných teplot budeme pracovat jen v režimu čtení. Naši úlohu tedy nebudeme řešit tak, že bychom si nejprve vyrobili kopii tohoto vstupního pole a nějakým způsobem v něm jednotlivé hodnoty upravovali nebo jinak předzpracovávali. Jinými slovy obsah a význam vstupních dat měnit nechceme, jen je chceme konkrétním způsobem použít a zpracovat.
118. [**Typy konstant**] Ve vztahu ke konstantám dodejme, že konstrukty `const` a `constexpr` se technicky liší. Zatímco v prvním případě jde v zásadě jen o obyčejnou proměnnou, jejíž hodnotu nemůžeme modifikovat, ve druhém případě jde o hodnotu, která je konstantní dokonce už v době kompilace. Kdykoli to tedy bude možné, budeme využívat právě je, protože mohou vést k efektivnějšímu kódu.
119. [**Velikost pole**] Tradiční céčkové pole, které pro naše teploty používáme, samo o sobě nezná svoji velikost. To znamená, že tento údaj musíme vždy vhodně předávat společně s polem samotným.
120. [**Velikost pole teplot**] Současně platí, že oba tyto údaje ze své podstaty logicky patří k sobě. Měli bychom k nim tudíž přistupovat symetricky i z technického pohledu na věc, tedy pokud jde o kontext jejich definice (globální vs. lokální) nebo formu jejich konstantnosti.
121. [**Předání pole parametrem**] Přestože céčkové pole je vlastně jen ukazatelem na jeho první prvek, i tak je lepší jej v parametru nějaké funkce deklarovat jako `int values[]` a nikoli `int * values`. Na první pohled pak tudíž bude zřejmé, že opravdu chceme pracovat s polem a ne jen jednou samostatnou hodnotou.
122. [**Velikost pole v parametru**] Přestože je zdánlivě možné u pole v rámci popisu očekávaných parametrů funkce popsat jeho velikost, tedy uvést např. `int values[size]`, tato velikost bude ve skutečnosti zcela ignorována, a nejde ji tedy k tomuto účelu použít.
123. [**Kontrola mezí pole**] Kdykoli pracujeme s polem, je naší vlastní povinností zajistit, že přistupujeme jen k jeho existujícím prvkům. Jinými slovy pokusy o přístup na neplatné pozice před jeho začátkem nebo za jeho koncem za nás nikdo neohlídá a neošetří. V lepším případě pak taková chyba vede k okamžitému pádu celého programu (signál 11 *segmentation violation*).
124. [**Počet průchodů**] Vždy bychom se měli snažit zpracovávat vstupní data jen pomocí nezbytně nutného počtu průchodů, ideálně jednoho jediného. To v našem případě zjevně kvůli zarovnání grafu teplot možné nebude, i tak se ale zbytečných průchodů vyvarujeme.
125. [**Opakované výpočty**] Obdobně se vyvarujeme zbytečně opakovaných dílčích výpočtů, které by stačilo realizovat jen jednou, protože jsou deterministické, tedy nad stejnými vstupy vždy vracejí stejnou hodnotu.
126. [**Hodnota minimální teploty**] Při hledání minimální teploty za účelem zarovnání grafu teplot je vhodné dobře si rozmyslet, jakou výchozí (nebo jinými slovy nejhorší možnou) hodnotu předpokládá

samotné zadání. V žádném případě není možné si jen tak vymýšlet nějaké konstanty, které v zadání nejsou předpokládány, jakkoli by se nám mohly zdát dostatečně vhodné, zejména velké nebo malé.

127. [**Hodnota neplatné teploty**] Neplatná teplota je určena pojmenovanou konstantou `no_value`. Pokud jde o skutečnou hodnotu této konstanty, může být libovolná. Tedy například `-999`, ale klidně i jakákoli jiná, ať už záporná nebo kladná. Opět tedy nemůžeme pracovat s předpoklady, jejichž platnost nám nikdo explicitně nezaručil.
128. [**Uniformní zpracování teplot**] Ke každému prvku v poli teplot je vhodné přistupovat stejně, žádná z platných teplot přeci nemá nějaké výjimečné postavení. Speciálně např. jen proto, že je uvedena jako první v pořadí. Zpracování všech hodnot by tedy mělo být z hlediska kódu rovnocenné.
129. [**Minimální vs. nejmenší teplota**] Konečně poznamenejme, že minimální prvek je matematicky něco jiného než nejmenší prvek, takže bychom na to měli myslet při pojmenovávání.
130. [**Zarovnání grafu teplot**] Rozhraní každé funkce by mělo být navrženo tak, abychom při jejím volání očekávali předání jen relevantních vstupních parametrů. Zejména bychom tedy volající neměli nutit zjišťovat a předávat nám takové informace, které jsou ve skutečnosti nedílnou součástí právě námi řešeného úkolu, a měli bychom si je tedy spočítat sami.
131. [**Vypisování symbolů**] Je zřejmé, že při vypisování grafu teplot budeme často potřebovat vypisovat nejrůznější počty určitých symbolů. Přímou se tedy nabízí, abychom si za tímto účelem připravili nějakou pomocnou, dostatečně univerzální funkci a tu pak jen opakovaně používali.
132. [**Opakování podobného kódu**] Předchozí myšlenka vlastně není o ničem jiném, než že bychom se vždy měli snažit vyvarovat se zbytečného opakování podobných nebo dokonce stejných fragmentů kódu. Tedy pokud k tomu není nějaký zvláštní důvod, což v našem případě není.
133. [**Přehnané větvení a zanořování kódu**] Při vypisování jedné konkrétní teploty je vhodné vyvarovat se přílišného nadužívání podmíněných výrazů, natož komplikovaných nebo zanořovaných. Zvláště když to v tomto případě není potřeba. Lepší je tedy snažit se namísto větvení hlavního kódu očekávaný počet symbolů spíše spočítat pomocí nejrůznějších operací nebo třeba i ternárního operátoru.
134. [**Pomocné kulaté závorky**] Přestože to není vždy nutné, dílčí podmínky v komplikovanějších logických výrazech je obecně vhodnější obalovat pomocí kulatých závorek. Nejenom že se tím zvýší přehlednost kódu, současně ale také nebudeme muset přemýšlet o prioritách operátorů, které někdy nemusí být přímočaré.
135. [**Volání funkce s různými parametry**] Pokud chceme zavolat nějakou funkci, ovšem dle situace s různými hodnotami parametrů, je vhodnější větvit kód pro určení těchto parametrů než hlavní kód, a volat tak tuto funkci vícekrát. Pomoci nám opět může i ternární operátor.
136. [**Poslední platná teplota**] Při vypisování grafu teplot bude zjevně nutné vyrovnat se situací, kdy aktuální hodnota není platná (tedy že jde o `no_value`). V takovém případě by nebylo dobrým nápadem pokoušet se polem pohybovat v opačném směru a poslední předcházející platnou teplotu nějak zpětně zkoušet dohledávat. A tím spíše ne pomocí nějaké rekurzivní funkce.
137. [**Přenášení teploty přes iterace cyklu**] Pokud potřebujeme přes jednotlivé iterace cyklu přenášet nějaké hodnoty, omezíme se jen na nezbytně nutnou informaci odpovídající logické podstatě řešeného problému. Nebudeme tedy přenášet žádné dopočítané, technické nebo pomocné údaje, ale opravdu jen teplotu jako takovou.
138. [**Obecné zásady kvality**] Případné komplikovanější části kódu doplníme o stručné komentáře. Samozřejmostí je dodržování všech obecných zásad kvalitního kódu. To například znamená, že bychom kód neměli psát pro sebe, ale pro někoho jiného. Tedy vytvářet takový kód, který bude srozumitelný i pro ostatní. Včetně nás samotných za několik let, kdy si už nepochybně nic pamatovat nebudeme.
139. [**Startovací balíčky**] Startovací balíčky k jednotlivým úkolům dostupné v rámci `ReCodExu` nejsou zamýšleny pro jejich přímé využití nebo začlenění v našem kódu. Slouží jen pro základní orientaci nebo vysvětlení některých technických nebo dalších aspektů.

140. [**Nedovolené prostředky jazyka**] V rámci úkolu se vyvarujeme použití jakýchkoli pokročilých prostředků, které jsme se neučili. Především nebudeme používat řetězce `std::string`, knihovnu `iostream` apod. Nechceme se totiž vzdálit od našeho záměru nízkoúrovňového programování. Navíc v dalších úkolech budou naše hardwarové prostředky při práci s Arduinem omezené, a tak se budeme vždy snažit o adekvátně jednoduchý a efektivní kód.
141. [**Kompilační varování**] Přestože kompilační varování na rozdíl od kompilačních chyb nemusí vždy indikovat chybný kód, jejich přítomnost ve většině případů skutečně signalizuje minimálně potenciální problémy nebo nevhodný kód. Z tohoto důvodu všechna taková případná varování také vyřešíme.