Hlutapróf 2

1. **Krossaspurningar (25%)**
   Staðsetning: Hlekkurinn Exam á skjáborði tölvunnar

2. **DoublyLinkedList: Insert og Remove (25%)**
   Staðsetning: Í möppunni Exam á skjáborði vélarinnar: Mappan ***StringList***
   Gefin er útfærsla á tvítengdum lista sem geymir strengi.
   Verkefnið er að fylla inn í útfærsluna á **`insert`** og **`remove`**.
   Implement both insert() and remove() using an implementation with sentinel nodes.
   Útfærslan notar “sentinel nodes”, þannig að head og tail eru ekki eiginlegir gagnahnútar heldur benda á fremsta og aftasta gagnahnút. Því á ekki að þurfa nein sértilfelli vegna tómra lista.

3. **Templates: Pair (25%)**
   Staðsetning: Í möppunni Exam á skjáborði vélarinnar: Mappan ***TemplatePair***
   Gefið er main fall sem býr til mörg mismunandi tilvik af klasanum Pair. Grind að klasanum Pair er einnig gefin, en þar á eftir að útbúa færibreytusmið, private gagnabreytur og setja klasann þannig upp að hann geti tekið við mismunandi týpum af göngnum í báðar gagnabreyturnar sem hann geymir. Auk þess á eftir að gera straumvirkjann << þannig úr garði að hann skrifi út gögnin á því formi sem “expected output” segir til um.
   Finish implementing the class Pair using templates so that it can take two variables of any type and the stream operator << so that each output matches the expected output.
   Klárið að útfæra klasann Pair þannig að main fallið þýðist óbreytt og straumvirkinn << skrifi gögnin út á réttu formi. Til þess þarf að ganga úr skugga um að klasinn innihaldi tvær gagnabreytur, sem hvor um sig getur verið af hvaða tagi sem sagt er til um þegar tilvik er búið til. Það þarf auk þess að útbúa færibreytusmið sem tekur inn þessi gögn.

4. **Recursion**
   Part 1: **Recursive multiplication (15%)**
   Staðsetning: Í möppunni Exam á skjáborði vélarinnar: Mappan ***RecursiveMultiplication***
   Notið endurkvæma forritun til þess að útfæra fallið int multiply(int a, int b). Fallið skilar margfeldi talnanna a og b, sambærilegt við (a * b). Þér er einvörðungu heimilt að nota reiknivirkjana '+' og '-' en EKKI '*' og '/'. ***Ekki þarf að gera ráð fyrir neikvæðum tölum.***
   *Use recursion to implement the function int multiply(int a, int b). The function returns the value of a and b multiplied together, equivalent to (a * b). You are only allowed to use the mathematical operators '+' and '-', but NOT '*' and '/'.* **You do not need to implement the solution for negative numbers.**

   Part 2: **X-ish (10%)** - *see next pages*

Staðsetning: Í möppunni Exam á skjáborði vélarinnar: Mappan ***RecursionXish***

Notið endurkvæmni til að útfæra fallið X-ish, **bool Xish(char* strX, char* strWord)**. Fallið skilar **true** ef orðið í **strWord** er "X-ish", skv. eftirfarandi skilgreiningu:

- Orð er "X-ish" ef það inniheldur alla stafina í orðinu X.

Hér þarf *ekki* að hafa áhyggjur af *hástöfum og lágstöfum*, einungis að skila **true** ef allir stafirnir, *nákvæmlega eins og þeir eru í strengnum **strX***, eru e-s staðar í strengnum **strWord**.

Use recursion to implement the function X-ish: **bool Xish(char* strX, char* strWord)**. The function returns **true** if the word in **strWord** is X-ish, based on the following definition:

- A word is X-ish if it contains all the letters in the string X

Here you do *not* need to *worry about upper and lower case*, only return **true** if *all the exact characters* in the string **strX** are also somewhere in the string **strWord**.

Alveg endurkvæm lausn (engar **for** eða **while** setningar) gefur full stig, *10*. Rétt lausn sem notar bæði endurkvæmni og lykkjur fær í mesta lagi *8* stig en lausn án endurkvæmni fær í mesta lagi *4* stig.

A fully recursive solution (no **for** or **while** statements) will give full points, *10*, a correct solution that uses both recursion and a loop will give *8* points at most, but a non-recursive solution, using only loops will give *4* points at most.

Mundu að þú mátt búa til hvaða hjálparföll sem henta, t.d. að búa til sér endurkvæmnisfall sem aðalfallið kallar í, ef þú vilt hafa aðgang að fleiri færibreytum en aðalfallið býður upp á. Það má líka breyta skilgreiningunni á aðalfallinu, en þó einungis þannig að main() fallið kalli rétt á þau án nokkurra breytinga á main() og check föllunum.

Remember that you are free to make any helper functions that you like, like making a separate recursive function called by the base functions, maybe to have access to more function variables than in the base functions. You can also change the base functions in any way, just making sure that the main() function can call them without any modifications to the code in the main function or the check functions.

Nánari ábendingar um útfærslu á næstu síðu.
More implementation tips on next page.

Að nota strenginn, char* sem lista af char:
Mundu einnig að **char*** er einfaldlega *kviklegt fylki af char*. Þú getur notað þetta eins og **lista**, með aðgang að fremsta stakinu og afgangnum af listanum. Til að kíkja á fremsta gildið í listanum er hægt að nota hvort sem er ***strWord** (það sem bendirinn bendir á, fremsta stakið) eða **strWord[0]**. Síðasti stafurinn í strengnum verður alltaf **'\0'**.
Til þess að vinna með lista sem byrjar á næsta staki á eftir núverandi fremsta staki er hægt að nota hvort sem er **(strWord+1)** (færir bendinn fram einn í listanum) eða **&(strWord[1])** (bendir á næsta stak í listnum). Bæði gera nákvæmlega það sama, bara mismunandi syntax. Þetta skilar líka **char***, sem er *listi af char* (eða strengur) sem *byrjar einu staki aftar í listanum*. Hugsað á sama hátt og *node->next*.

Using the string, char* as a list of char:
Also remember that the **char*** that is sent into your functions is simply *a dynamic array of **char***. You can view this as a **list**. In order to check the *first item* in that list you can dereference the pointer, either with ***strWord** (what the pointer points to, the first character) or **strWord[0]**. The **last char** in each string will always be **'\0'**.
In order to access a list that starts at the next item after the current char* you can use **(strWord+1)** (moves the pointer forward one item in the list) or **&(strWord[1])** (a pointer to the next item in the list). Both are equivalent, only differ in syntax. This is a **char*** that is also a list of **char**, or a string, *that starts one item later in the list*. Think of it like *node->next*.

Fyrstu prófanirnar í *main()* nota allar orðið *"elf"* sem *X*.
Það má notfæra sér þetta til að útfæra fyrst einfaldaða útgáfu, *elfish*, sem athugar bara hvort strengurinn word inniheldur stafina ***e***, ***l*** og ***f***. Síðan má bæta við útfærsluna til að tékka á stöfunum í hvaða streng *X* sem er. Veltið líka vel fyrir ykkur hvorn strenginn er betra að fara gegnum sem *ytri* "lykkju" og hvorn sem *innri* "lykkju".

The first tests in *main()* all use the word *"elf"* as *X*.
This can be used to first implement a simplified version, elfish, that only checks whether the string word contains the lettters ***e***, ***l*** and ***f***. Then this can be extended to check for the letters in any string *X*. Also think about which string it makes more sense to go through in an *outer* "loop" and which one in the *inner* "loop".