Laboration 1: ADTer, Dynamiskt minne, pekare och rekursion

Uppgift 1: ADTn DString

Vi har designat en ADT som heter `DString`. Datatypen är till för att hantera dynamiskt allokerade strängar. Datatypen innehåller följande operationer:

- Initialize: Vi kan skapa en ny dynamisk sträng.
- Concatenate: Vi kan sammanföra två dynamiska strängar till en.
- Truncate: Vi kan korta ner en dynamisk sträng.
- Print: Vi kan skriva ut en dynamisk sträng på skärmen.
- Delete: Vi kan radera och avallokera minnet för en dynamisk sträng.

Det finns på Canvas ett “labbskelett” som innehåller interfacet för denna ADT. Din uppgift kommer att vara att implementera funktionaliteten som behövs för att realisera denna ADT.

Uppgift 1.1: Kompilera labbskelettet

Ladda ner labskelettet “dstring.zip” från Canvas. Öppna ett nytt projekt som ni kallar för “dstring” och lägg till alla filer från labskelettet till ert projekt. Kontrollera att ni kan kompilera projektet utan några felmeddelanden (det kan dyka upp varningar - ignorera isåfall dem just nu). Om ni försöker köra programmet kommer det dock att krasha - vilket är meningen.

Uppgift 1.2: Bekanta dig med ADTn `DString`

Filen `dstring.h` innehåller interfacedelen av `DString` som är en ADT för dynamiskt allokerade textsträngar. Den innehåller fem stycken funktionsdeklarationer. Med dessa funktioner kan man initera/allokera, konkatenera (slå ihop), trunkera (förkorta), skriva till fil samt frigöra minnet för en dynamisk sträng. Filen innehåller även kommentarer som beskriver vad funktionerna gör.

Filen `dstring_test.c` innehåller ett program som testar att ADTn fungerar. Studera denna fil noggrant och försök få en förståelse för hur ADTn är tänkt att användas. När uppgift 1.3 är klar ska det gå att köra detta program utan några fel.

Filen `dstring.c` är ofärdig. Denna kommer att innehålla implementationsdelen av ADTn. Det är er uppgift att implementera dessa funktioner.

Innan ni går vidare till uppgift 1.3, se till att ni förstår vad alla funktioner i `dstring.h` är tänkta att göra. Diskutera gärna med labbassistenten för att se till att ni förstått rätt.

Uppgift 1.3: Implementera ADTn `DString`

Det är nu dags att implementera de fem funktionerna för `DString` i `dstring.c`. Sätt asserts för pre- och postconditions efter kommentarerna i c-filen. C-filen innehåller även några tips på vad ni bör tänka på.

Notera att typen `DString` är definierad (via `typedef`) som en **char**-array: dvs en vanlig sträng. Eftersom arrayer fungerar som pekare så får alltså typen `DString` typen **`char`***. Detta betyder att typen `DString`* har typen **`char`****, dvs pekare-till-pekare. Detta är nödvändigt för att man ska kunna skicka in en pekare till en sträng för att funktionen ska kunna ändra det pekaren pekar på (t.ex när den allokerar dynamiskt minne).

Ni får inte göra några ändringar i `dstring.h` eller `dstring_test.c`! Ni får enbart ändra i `dstring.c`!

Uppgift 1.4: Testa implementationen

Nu när ni skrivit klart alla funktioner är det dags att se att er ADT fungerar. Provkör programmet i `dstring_test.c`. Allt fungerar bra om allt kompilerar och att programmet avslutas. Programmet ska även skriva ut texten “Department of Redundancy Department” på skärmen om allt fungerar som det ska. Om något annat händer så får ni undersöka var det har blivit fel. Förslagsvis genom att stega igenom steg-för-steg med debuggern.

Tips: Det kan vara bra att testa funktionerna allteftersom de blir färdiga, det går då bra att använda testfilen men kommentera bort de rader som rör funktioner som ännu inte är färdiga.

Uppgift 2: Rekursion

Skriv en funktion `sum()` som rekursivt beräknar 1+2+3+4+...+*n* för ett angivet tal $n \geq 1$. Om man anropar funktionen med t.ex 3 ska alltså talet 6 (1+2+3) returneras.

För att öka förståelsen för hur rekursiva funktioner fungerar ska funktionen även skriva ut en spårutskrift på skärmen. I början av funktionen ska ni skriva vilket tal funktionen anropades med, och i slutet av funktionen ska ni skriva ut vilket tal som returneras.

Utskriften för `sum(3)` bör se ut något i den här stilen:

```
sum(3) anropas
sum(2) anropas
sum(1) anropas
sum(1) returnerar 1
sum(2) returnerar 3
sum(3) returnerar 6
```

Lägg till följande följande asserts i `main()` för att testa funktionen (för att asserts ska fungera måste biblioteket `assert.h` vara inkluderat):

```
assert(sum(1) == 1);
assert(sum(2) == 3);
assert(sum(3) == 6);
assert(sum(4) == 10);
assert(sum(5) == 15);
assert(sum(20) == 210);
assert(sum(0) == 0);
assert(sum(-1) == 0);
```

Titta också på utskrifterna för att se att du säkert fått med alla anrop och returer.