NHF (Szemétgyűjtő)
Kész

Feladat:

Szemétgyűjtő

Tervezzen olyan segédobjektumokat, amelyek lehetőséget adnak az automatikus "szemétgyűjtésre" (a már nem használt dinamikus memóriaterületek automatikusan felszabadulnak)!

A működés alapja röviden a következő: Nyilván kell tartani minden dinamikusan létrehozott objektumban, hogy hivatkoznak-e rá pointerrel. Ha nem, akkor az objektum törölhető a dinamikus memóriából. Ez megvalósítható, ha minden objektumot egy speciális alaposztályból származtatunk, valamint minden objektumot csak egy ún. "okos" (smart) pointeren keresztül érünk el, ami természetesen szintén egy objektum.

Specifikáljon egy egyszerű tesztfeladatot, amiben fel tudja használni az elkészített adatszerkezetet! A tesztprogramot külön modulként fordított programmal oldja meg! A megoldáshoz ne használjon STL tárolót!

Pontosított specifikáció:

A feladat célja:

A feladat olyan segédobjektumok tervezése, melyek lehetőséget adnak automatikus szemétgyűjtésre, ezzel megkönnyítve a programozó dolgát.

Az automatikus szemétgyűjtő használata:

A szemétgyűjtő segítségével írt programban a „hagyományos” pointerek helyét átveszik az úgynevezett okos pointerek, melyek képesek nyilvántartani, hogy egy adott dinamikusan foglalt objektumra történik-e hivatkozás, vagy már nem. Amennyiben több hivatkozás nem mutat az adott memóriaterületre, a szemétgyűjtő ezt felismeri, majd felszabadítja a lefoglalt memóriaterületet.

Az okos pointerek ugyanúgy használhatóak, mint hagyományos társaik, az operátorok ugyanúgy hatnak rájuk.

A szemétgyűjtő aktiválásához elegendő a megfelelő header fájl beillesztése, illetve a dinamikusan foglalt memóriaterületekre mutató pointerek lecserélése okos társaikra.

Az okos pointereket ugyanúgy konstansnak jelölhetjük a `const` keyword használatával. Ezáltal csak olvashatóvá tehetjük a tárolt pointert és/vagy a pointer által mutatott memóriaterületet.

A szemétgyűjtő tesztelése:

A végleges program tartalmazni fog egy tesztfeladatot, amely demonstrálja az elkészített szemétgyűjtő működését, illetve bemutatja a túlterhelt operátorokat is. A memóriaszivárgásokat a tesztprogram a memtrace környezettel fogja tesztelni.

A tesztfeladat tartalmazni fog egy dinamikusan foglalt láncolt listát, egy dinamikusan foglalt mátrixot, illetve tesztelni fogja a megvalósított operátorokat is.

Hibakezelés:

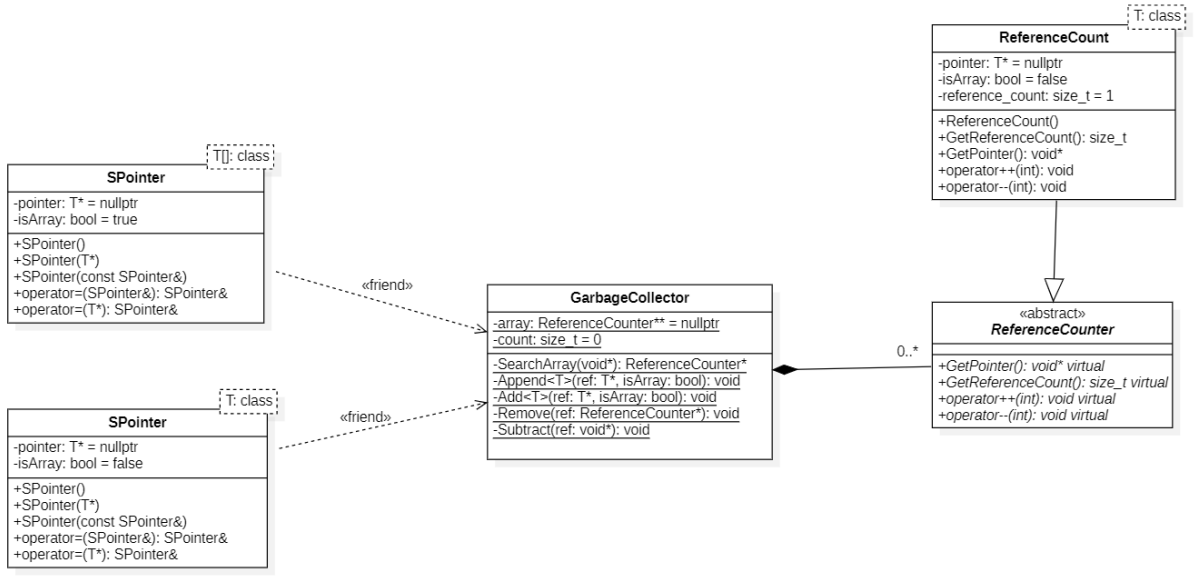
Amennyiben egy új memóriaterület lefoglalásakor a szemétgyűjtő hibát tapasztal, const char* típusú kivételt dob, és felszabadítja a lefoglalt memóriaterületet.

Ilyen hiba keletkezhet például amikor a szemétgyűjtő a referenciák számát nyilvántartó tömböt próbálja átméretezni.

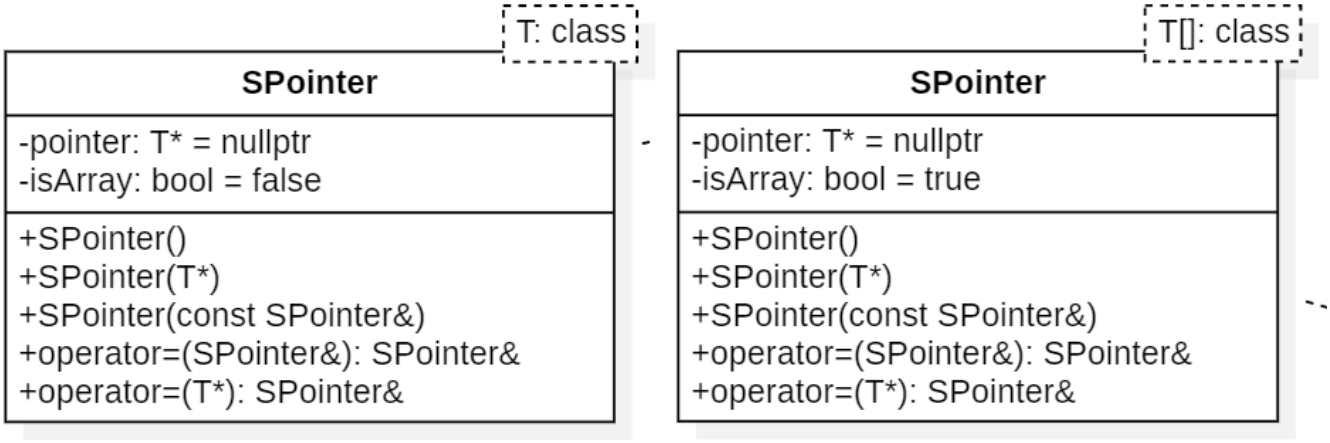
A szemétgyűjtő szintén const char* hibát dob, ha olyan memóriacímhez tartozó referencia-számláló csökkentésére történik kísérlet, ami nem létezik.

Terv:

Osztálydiagram:

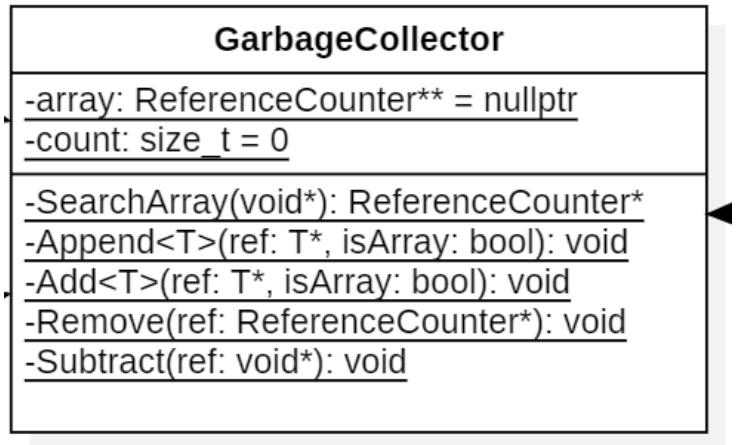


SPointer:



- Ez az osztály maga az okos pointer, melyet a hagyományos pointerek helyett lehet használni dinamikusan foglalt memóriaterület esetén.
- Amennyiben az okos pointerben dinamikus tömbre mutató pointert szeretnénk tárolni, T[] formátumban kell létrehozni a templatet.
  - Pl: `SPointer<int[]> array = new int[42];`
  - Mivel az értékadó operátor lehetőséget ad a tárolt pointer kicserélésére, ezért figyelni kell rá, hogy ne tároljunk el sima változóra mutató pointert tömb típusú okos pointerben, illetve egyanez igaz visszafelé is. Mivel a program a két pointer típust nem tudja megkülönböztetni, ez a feladat a programozóra hárul.

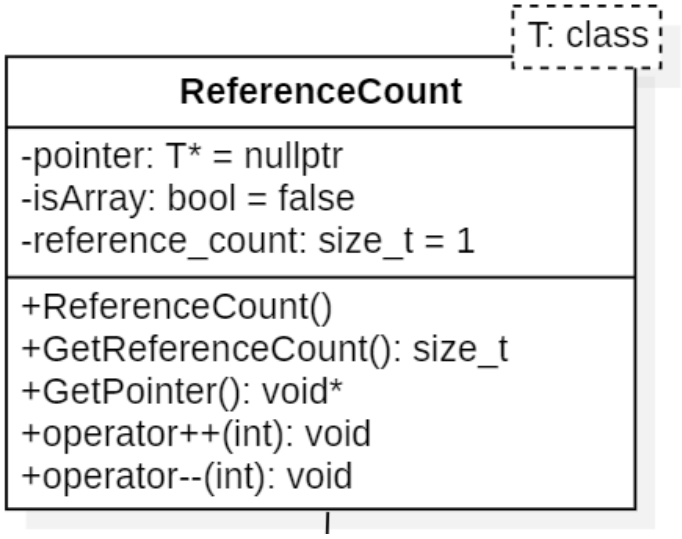
GarbageCollector:



- Ez az osztály tartalmazza magát a szemétgyűjtőt. Az osztály minden változója és függvénye statikus. Az osztály függvényeit kívülről csak az okos pointerek érik el (SPointer).
- Az osztály egy dinamikus tömbben (heterogén kollekció) tárolja az összes dinamikusan foglalt memóriaterületre mutató pointert, illetve a rájuk hivatkozó okos pointerek számát. Ha minden pointer felszabadításra kerül, a dinamikus tömb is felszabadul.

- Az okos pointerek által használt két függvény az Add és a Subtract:
- Add:
    - A függvény megkapja az okos pointer által tárolt pointert, illetve információt kap róla, hogy a pointer tömbre mutat-e, vagy sem.
        1. Meghívódik a SearchArray függvény, ami megnézi, hogy az adott pointerre létezik-e már referencia.
        2. a) Amennyiben igen: eggyel növeli a pointerre mutató referenciák számát.
           b) Amennyiben nem: létrehoz egy új referencia-számlálót a tömbben (Append).
- Subtract:
    - A függvény megkapja az okos pointer által tárolt pointert
        1. Meghívódik a SearchArray függvény, ami megnézi, hogy az adott pointerre létezik-e referencia.
        2. a) Amennyiben igen: Levon egyet a pointer referencia-számlálójából. Amennyiben a számláló 0 lesz a kivonás után, a memóriaterület felszabadításra kerül, és a referencia-számláló törlődik a tömbből (Remove).
           b) Amennyiben nem: const char* kivétel dobódik.

ReferenceCount:



- Ez az objektum tartja számon, hogy hány darab okos pointer mutat egy adott memóriacímre. Ha az érték 0, felszabadítja a memóriaterületet.
- Leszármazottja a *ReferenceCounter* absztrakt osztálynak, mely lehetővé teszi egy heterogén kollekció létrehozását.

Tesztek:

A legtöbb teszt a memtrace könyvtárra hagyatkozik, hiszen, ha a program futásának végén nem jelez szivárgást, akkor biztosak lehetünk benne, hogy jól működik a szemétgyűjtő.

Értékadás teszt:

Teszteli, hogy az okos pointer objektum használható-e sima pointerek helyett dinamikusan foglalt címek esetén. Lefoglal egy új memóriaterületet, majd beállít neki egy értéket. A teszt végén a lefoglalt memóriaterület felszabadításra kerül.

Láncolt lista teszt:

Létrehoz egy 5 elemből álló láncolt listát, majd elkezd törölni a végéről elemeket. A törölt elemek automatikusan felszabadításra kerülnek. A teszt végén a még nem törölt elemek is felszabadulnak.

Mátrix teszt:

Létrehoz egy 10x10-es mátrixot, majd feltölti számokkal 0-tól 99-ig. A teszt végén a teljes mátrix automatikusan felszabaditásra kerül.

Dokumentáció:

A program részletes dokumentációja ezen a linken található, doxygen által készített html formátumban.