1. géptermi gyakorlat – (ezazamaz + vektor)

Néhány fontos programozási „cucc” `C++` szemmel

Enkapszuláció (egységbe zárás):

Egy olyan programozói alapelv, amit akkor követünk, amikor egy bizonyos összetett megvalósítású egység (*függvény*, *osztály*, stb) belső logikájának részleteit elrejtjük es annak használója felé csak bizonyos szolgáltatásokat adunk, amelyeket anélkül is tud használni (pl. mert leírtuk a dokumentációban!), hogy átnézné részletesen es megértené a megvalósítás kódját

Blokk:

`C++`-ban a `{}` közötti utasítások alkotnak egy blokkot. Minden a blokkban deklarált *statikus* memóriafoglalású változó a blokk végéig él

Élettartam (lifetime, lifecycle):

Egy változóhoz ennek idejéig tartozik memóriaterület.
Statikus változók: a szülő blokk végéig,
globális változók: a program futásának végéig,
dinamikus változók: a pointer által mutatott terület explicit lefoglalásától felszabadításáig vagy a program futásának végéig tart

Hatókör (scope):

A programszöveg azon része, ahol az adott változóra, függvényre valahogyan hivatkozni tudunk.
Statikus: az élettartama alatt,
globális: végig,
dinamikus: amíg az ún. *pointer* erre a memóriacímre mutat. függetlenül attól, hogy tényleg ott van-e az adat. Ezért a dinamikus hatókör lehet szűkebb és bővebb is, mint az élettartam

Láthatóság (visibility):

A programszöveg azon része, ahol az adott változóra, függvényre az adott névvel tudunk hivatkozni.

Elfedés (shadowing, hiding) történik akkor, amikor egy blokkban egy az ő szempontjából külső blokk már deklarált változójával megegyező nevű változót deklarálunk.

Ekkor a belső blokkban mindkettő változó életben van, de a külsőt vagy csak *minősített névv*el vagy sehogy se tudjuk elérni, mert a belső „elfedi” a külsőt.

A blokk végén a belső meghal, így a név megint a külsőt illeti meg.

Azonos blokkban több azonos nevű változót nem lehet deklarálni különböző típusozással sem.

Az elfedés adja a *hatókör* és a *láthatóság* különbségét

Minősítés (identifier, specifier):

Minősített névvel érhetjük el az osztály vagy névtér elfedett változóit, ehhez a *scope resolution operátor*t (`C++`-ban `::`) használhatjuk.

Névtér (`namespace`):

Változókat, osztályokat, függvényeket, stb-ket össze tudunk gyűjteni egy ilyen logikai egységbe, ha ezek után hivatkozni szeretnénk a névtér elemeire vagy minősíteni kell őket, vagy előbb ki kell adni a `using namespace …` utasítást. Névkonfliktusokat lehet használatával megelőzni.

Sablon (`template`):

Egy olyan osztály, ami egy általános *adatszerkezet*et vagy egy olyan függvény, ami egy általános *algoritmus*t fogalmaz meg, tekintet nélkül arra, hogy pontosan milyen típusú elemekkel dolgozunk. Ez által el tudjuk érni, hogy a hasonló algoritmusokat elég legyen egyszer megírni, majd később ezt a sablont kitöltve „példányosíthassuk” a konkrét változatokat, akár olyan típussal is, ami mondjuk a sablon megírásának pillanatában még nem is létezett.

Statikus és dinamikus memóriakezelés:

A különbség lényege, hogy a statikus esetben már *fordítási idő*ben tudnunk kell a konkrét memóriafoglalási igényeinket (azaz, mennyi memóriát kell foglalnunk és ebből mennyi az egybefüggő (*tömb*ök)), dinamikus esetben pedig futás időben, a felhasználó által beírt értékek alapján foglalunk – mindig csak annyit, amennyit kell.

A statikus memórián létrejön egy *mutató* (*pointer*). Ez gyakorlatilag egy szám, egy memóriarész címkéje. Ez a memóriacím már a dinamikus memórián van, azt sosem direktben, hanem a pointeren keresztül érjük el. A pointert átállíthatjuk más címre, a cím által mutatott részt lefoglalhatjuk, felszabadíthatjuk, lekérdezhetjük, felülírhatjuk. A címet sose mi mondjuk meg, hanem az operációs rendszer osztja ki, igényeinknek megfelelően.

Előnye a dinamikus változatnak, hogy nagy méretű adatszerkezetek esetén nem pazarlunk annyi memóriát, részint mert annyit foglalunk, amennyit kell, részint mert bármikor (azaz akár használat után közvetlenül) fel tudjuk szabadítani. Hátránya kis méretű adatoknál, hogy mégiscsak több memóriát foglalunk, a pointer miatt, ami maga egy sima statikus változó! További hátránya, hogy bonyolultabb átlátni, illetve, hogy mi vagyunk felelősek a változó élettartamáért. Ha hibát követünk el, a fordító nem valamilyen szép, többnyire érthető hibaüzenettel adja ezt tudtunkra, amit emiatt könnyen ki tudunk javítani, hanem egyszerűen csak futás közben elszáll a program. Ráadásul ha a hiba ritka, megeshet, hogy a program kiadása után jövünk rá.

A pointerek nem részei az idei féléves anyagnak. Egyébként a `C++`-ban a `*` karakter jelöli a pointert. Az általunk idén használt vektorok is dinamikus memóriában tárolják az adatokat, de mindez el van előlünk rejtve, mi a vektor típusnak csak „felhasználói” vagyunk, magukat a vektorokat statikus változókként kezeljük, ezért nem látunk sehol `*`-ot.