**Magasszintű Programozási Nyelvek I Vizsga jegyzet**

Ez a jegyzet nem gyakorlati, arra ott voltak a drillek, meg a gyakorlás, ez sokkal inkább egy elméleti, szemléleti dolog lenne, de ebből valószínűleg az a clusterfuck lesz amit Stroustop összehordott lmao

**Chapter 12. – A Display Model**

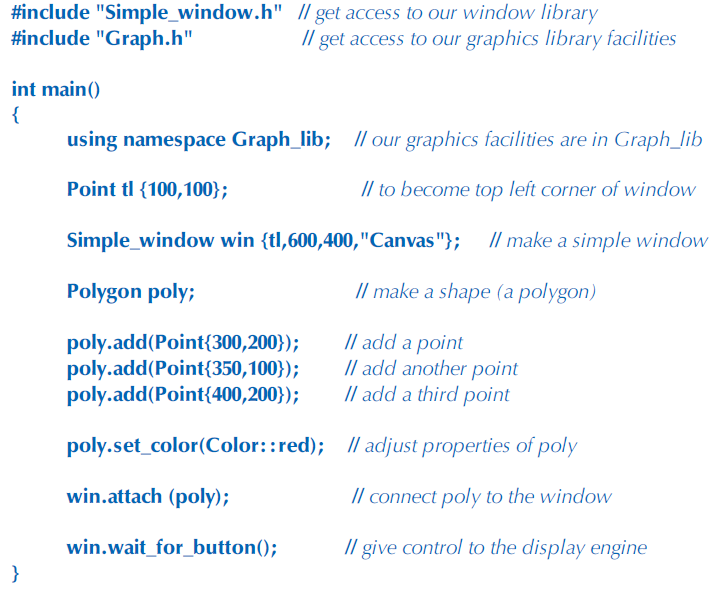
**Miért foglalkozunk Grafikával?**

* Vizualizálás, legfőképp tudományos szférában, illetve általánosan adatok vizualizálása
* Alapvető grafikai dolgok bemutatása
* Példák bemutatása, melyek nehézségei, megoldásai jók lehetnek más programok tervezése esetén
* Ezen keresztül mutatja meg a könyv az Objektum Orientált Programozást

**Alapvető modell**

A HTML-t mutatja be a könyv, mint példa, ahol elég könnyen tudunk grafikai dolgot készíteni, de annak megvannak a határai.   
A modell, amit mi használunk:

* Csinálunk alapvető objektumokat, amiket – jelen esetben – az FLTK biztosít
* Ezeket a dolgokat „felcsatoljuk” egy ablak objektumra, ami azt reprezentálja, amit majd a fizikai képernyőn látunk
* Az FLTK, a mi GUI Librarynk fogja az objektumokat, amiket felraktunk az ablakra, és kirajzolja őket a képernyőre
* Lényegében a Display Engine rajzol ki mindent, erre GUI Libraryként fogunk hivatkozni

**Bevezető Példa**

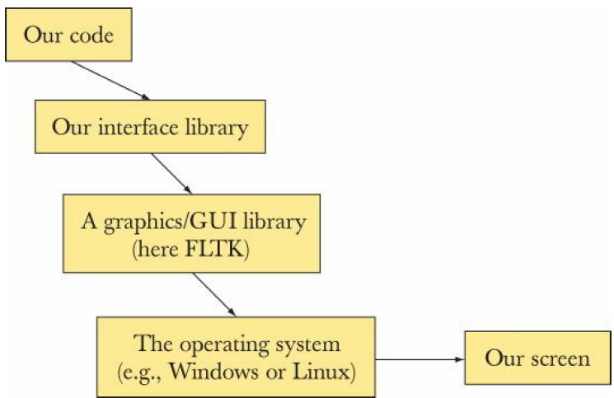
Itt mi a fontos?

Includolva kell a két header, a Simple\_Window-ban van az ablak, a Graph.h-val jön lényegében az egész FLTK, anélkül nem nagyon működik semmi

A Using Namespace csak egy kényelmi dolog, hogy ne kelljen mindenhova odaírni, hogy Graph\_lib::

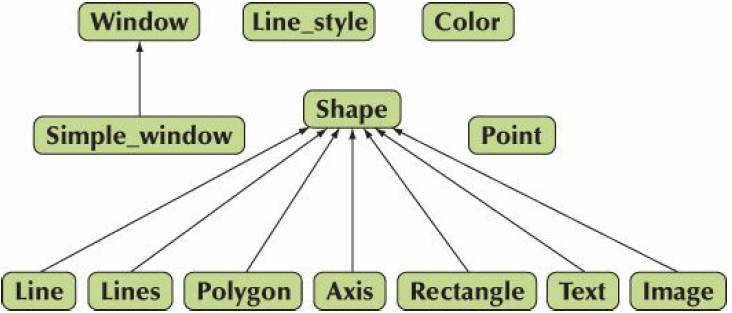
Látható az alap modell, csinálunk objektumokat, mint a Pont, a sokszög, és ezeket felcsatoljuk az ablakra.

Minden objektum lényegében egy osztály, és az összes alakzat a Shape ősosztályból eredeztetett.

**GUI Library-k használata, miért FLTK?**

Használhatnánk az Oprendszer beépített GUI Library-jét, azonban azzal limitálnánk magunkat egy Oprendszerre, and that we no likey

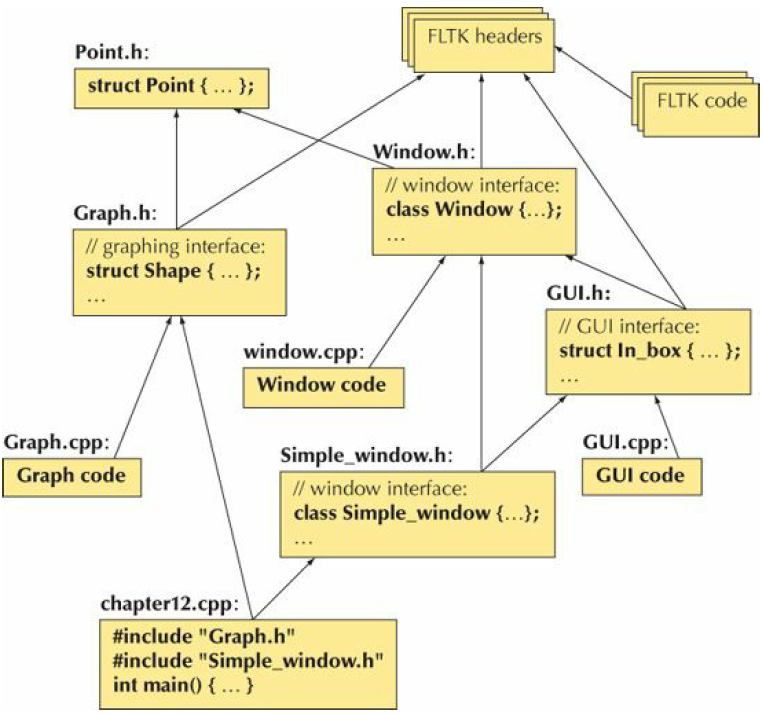
A C++-nak nincs egy standard GUI Library-je, ahogy van egy I/O Library-je, ezért kell egy c++ GUI Library amit használhatunk 🡪 FLTK

**Az alapkő – Shape**

Később ezt részletesen leírja a könyv, itt csak bemutatja, miként az alapja minden alakzatnak a Shape, azaz a Shape egy ősosztály. Minden objektum, amit fel tudunk csatolni az ablakra, egy Shape-ből eredeztett osztály, objektum.

**Rendelkezésünkre álló objektumok**

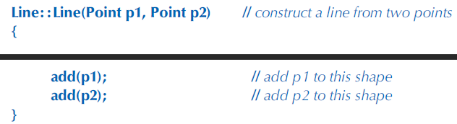
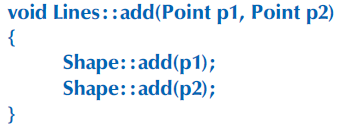
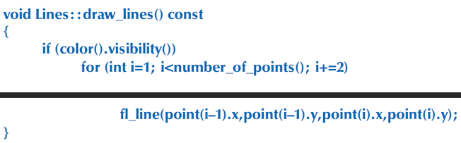
* Axis
* Function
* Polygon
* Rectangle
* Text
* Open/Closed Polyline
* Image

** Hogy is működik együtt az FLTK, és minden más?**

**Chapter 13. – Grafikai Osztályok**

Rengeteg grafikai osztály van az FLTK-ban, de mi csak egy párral fogunk foglalkozni, és azoknak a működése lesz nagyjából elmagyarázva ebben a fejezetben.

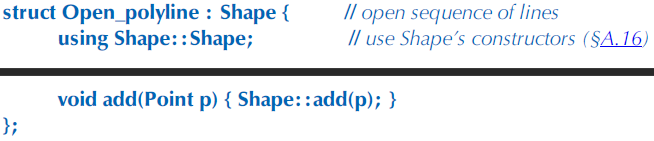
**Point és Line(s)**

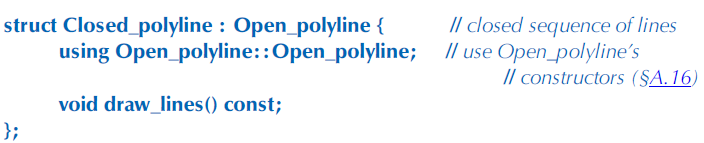
Minden alakzat alapja a pont, a vonal pedig lényegében 2 vagy több pont. Egy pontot két koordinátával(x,y) adunk meg. Mindkettő a bal felső sarokban 0, és onnan lefelé, illetve jobbra nő.  
Egy Pont lényegében egy osztály, két int-el, egyik az x másik az y koordinátát reprezentálja. Emellett van két operátor hozzá, == és !=  
A Line egy Shape-ből eredeztetett osztály, amit két Pont megadásával tudunk definiálni.  
Többféleképpen lehet a Line osztályon belül megadni a két pontot, de mivel a Shape-ből eredeztetett, ezért használhatjuk a Shape add() függvényét.   
A Lines objektum szintén egy Shape-ből eredeztetett osztály, ami lényegében Line objektumokat tud tárolni, tehát egyszerre több vonalat. Ennek az az előnye, hogy egyszerre tudunk végrehajtani műveleteket vonalakon, nem kell egyesével színt, stílust változtatni. A vonalakat a Lines-hoz szintén add()-al tudjuk hozzáadni. Itt látszik, hogy a Shape::add-ja az amit felhasználnuk.   
Az add által megadott pontokat a draw\_lines() függvény fogja megrajzolni. Azért const, mivel nem változtathatja meg magát a Shape-t.

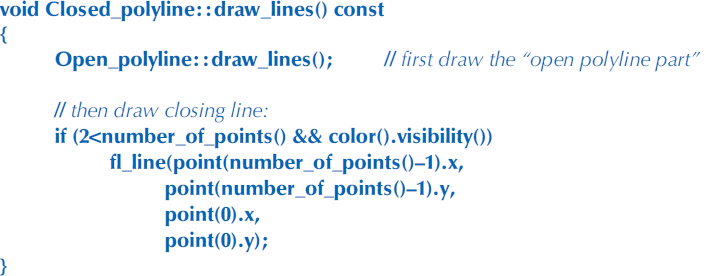
**Vonalstílus, Szín**

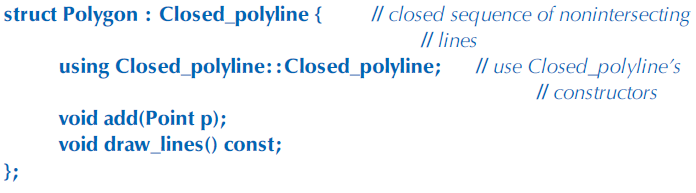
Az FLTK-hoz több szín, illetve vonalstílus tartozik, melyeket lényegében minden objektumra tudunk használni.

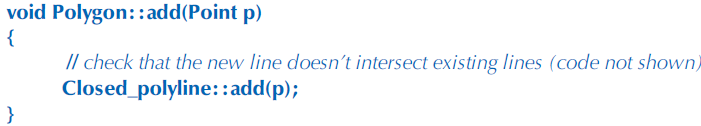
**Open Polyline**

Az Open Polyline egy Shape-ből eredeztett objektum, mely több vonalszegmens összekötve. Az Open Polyline megörökli Shape add()-ját, és a draw\_lines()-t is, mivel az Open Polyline nem áll másból, mint sok pontból, melyeket össze kell kötni. Ez az előnye az OOP-nak I guess.   
A Shape::Shape egy Using lényegében, hogy használhatja a Shape konstruktorait, az alap és a lista konstruktort, ami lényegében több add().

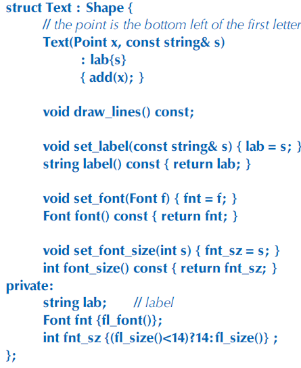
**Closed Polyine**

A Closed Polyline egy Open Polyline, amelynek az utolsó pontja össze van kötve az első ponttal. Itt jön be újra az OOP, és megkapja az Open Polyline konstruktorait, de mivel itt össze kell kötni az első és utolsó pontot, ezért saját draw\_lines() kell neki, amit szintén tudunk eredeztetni a az Open Polylines draw\_lines()-ból, hiszen azon felül kell még plusz egy vonalat rajzolnia.

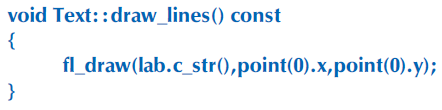
**Polygon**

Egy olyan Closed Polyline, ahol a vonalak nem keresztezhetik egymást. Tehát ezt tudjuk a Closed Polyline-ból eredeztetni, viszont minden egyes add()-ot ellenőriznünk kell, hogy keresztezik-e utána egymást vonalak. Ez nem egy hatékony algoritmus, de ez van .

**Rectangle**

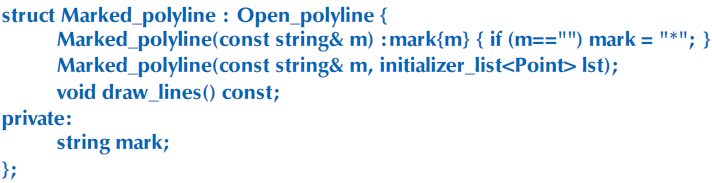
Egy téglalap mi a faszomat vártál lmao  
Meg lehet adni kétféleképpen🡪Bal Felső sarok, hossz, szélesség/Bal felső sarok, jobb alsó sarok  
Tehát 1 Pont, és 2 int, vagy 2 pont az, amivel meg lehet adni. Itt vannak még ilyen színezős dolgok, de idc 🡪 Kitöltő színezés, meg vonalszínezés

**Text**

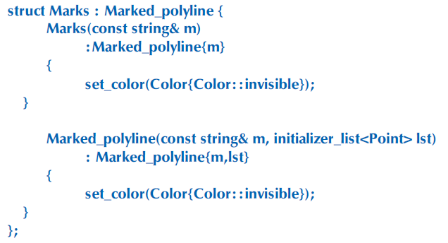
Shape-ből eredeztetett, meg kell adni egy pontot, ahol a szöveg bal alja fog kezdődni, illetve meg kell adni a stringet, hogy mi lesz maga a szöveg. Meg lehet adni stílust, méretet, színt. A text-nek saját draw\_line()-ja van, mivel elég máshogy kell rajzolni, mint egy alakzatot.

**Kör/Ellipszis**

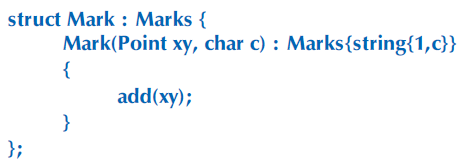
A kör is egy Shape, amit a középpontjával, és a sugarával tudunk megadni.  
Az Ellipszis is egy Shape, amit a középpontjával, valamint a magasságával, és szélességével adunk meg. Itt nem sok érdekes dolog van Tbh.

**Marked Polyline**

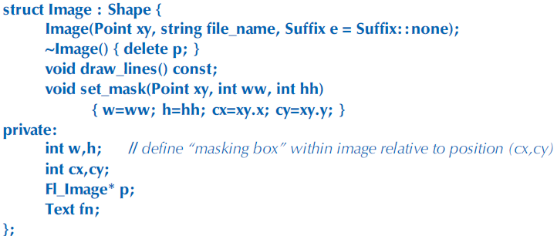
A Marked/Jelölt Polyline egy Open Polyline, aminek a pontjai valamivel jelölve vannak. Mivel meg kell rajzolni a jelöléseket, saját draw\_lines() kell neki.

**Marks**

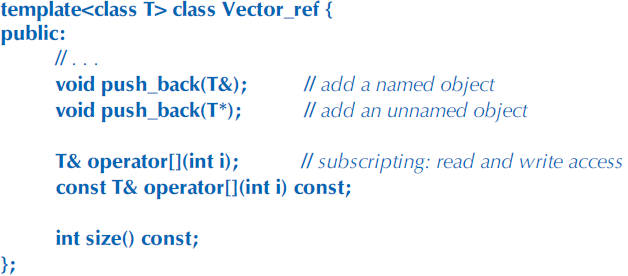
Jelölt pontok sorozata. Elég agyfasz módon, ez egy Jelölt Polyline, ahol a vonalak láthatatlanok. Mi a fasz

**Mark**

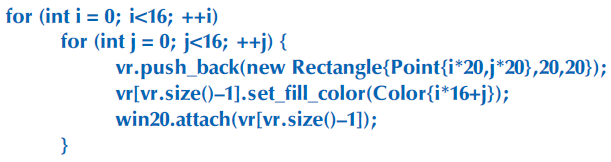
Egy darab jelölt pont. A Marks-ból eredeztetett, de egy darab csak.

**Image**

Minden kép is egy Shape, amit egy Ponttal lehet megadni, ami feltételezem a kép bal felső sarkát jelenti, illetve a képfájl nevével. Csak Jpg és gif lehet.

**Vector\_Ref – Névtelen Objektumok tárolása**

Egy generikus tároló, amiben meg van oldva a nevezett, és névtelen objektumok tárolása. Ez ebben a drillben hasznos volt, ebben kellett eltárolni az átlóba kerülő piros négyzeteket. Névtelen objektumokat a new kulcsszóval tudunk létrehozni, majd tárolni.

Példa:

**Chapter 14.  
Grafikai Osztályok Készítése – Absztrakt Osztályok**

**Koncepciók**

Egyszerű koncepciókat akarunk, amibe nehéz belezavarodni.

* Ablak 🡪 Ahogy az Oprendszer adja
* Vonal 🡪 Ahogy látjuk a képernyőn
* Pont 🡪 Egy koordinátapár
* Szín 🡪Ahogy látjuk a képernyőn
* Shape 🡪 Na ez mi?

A Shape egy absztrakt dolog. Soha nem látunk csak egy Shape-t. Egy fajta/féle Shape-t látunk.  
Az osztályokban, amiket eddig megnéztünk egy közös dolog van, mind egy Shape. Ez az alapvető koncepciónk, van egy ősosztály, a Shape, amiből tudunk mindent eredeztetni. Tehát amikről beszéltünk nem különálló random dolgok, hanem mindig van egy dolog, ami összeköti őket. Ezzel tudunk általánosítani. Meg lehetne írni minden külön, de mivel elég sok hasonlóság, és közös pont (pun intended) van az objektumokban, amikről beszélünk, ez nem lenne praktikus. A másik véglet, hogy egy Shape osztályba írunk meg mindent. Alapvetően nem értek hozzá én sem, de logikusan hangzik, hogy könnyebb több, kisebb valamilyen szinten összefüggő osztályt kezelni, mint egy nagy osztályt.

**Műveletek**

Minél kevesebb műveletet, akarunk minden osztályban, hogy annyira egyszerű legyen, amennyire csak lehet. Ennek egy példája, mondjuk az add() függvény, vagy az, hogy lényegében minden objektumunknak, ha egy pont kell a képernyőn, akkor a Point osztályt használjuk, az argumentumokat, ahol lehet ugyanabban a sorrendben kérjük, hogy minél átláthatóbb legyen. Használhatnánk Point objektumok helyett egy koordinátapárt, amikor pont kell egy objektumnak, de ez a konzisztencia, hogy mindenhova egy Point objektum kell leegyszerűsíti a dolgokat.

**Attach/Add közötti különbség**

Az add()-al hozzáadunk valamit egy objektumhoz, ez a valami legtöbbször egy pont, vagy egy vonal. Az add() után a objektum csinál egy saját példányt „magának” az adott dolgokból. Ezzel szemben az attach(), az csak egy referencia. Amikor felrakunk az ablakra egy objektumot, akkor az ablak nem csinál egy saját példányt, hanem csak egy referencia lesz arra az adott objektumra. Tehát ha az objektumot, amire hivatkozik az ablak kitöröljük, nem fog működni.

**A Shape – Tis gon be a long one**

Ahogy már eddig is beszéltünk róla, a Shape egy absztrakció, egy fogalom, arra ami megjelenhet a képernyőn. A Shape egy osztály végső soron, egy absztrakt osztály, ami foglalkozik azokkal a dolgokkal, amik közösek a legtöbb Shape objektum esetén, vonalszín, vonalstílus, kitöltési szín. Emellett pontokat is tud kezelni, hiszen minden Shape-ből eredeztetett objektumnak szüksége van pontokra. Emiatt van egy draw\_lines (), és egy add() függvénye, amikről már beszéltünk eddig is.

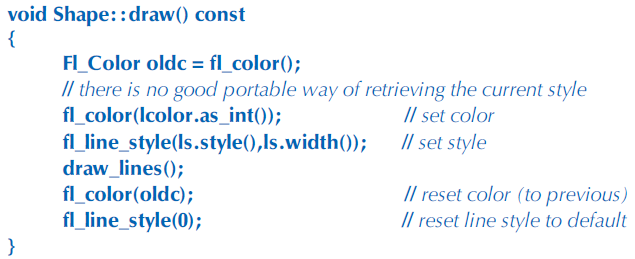
Honnan látjuk, hogy abszrakt egy osztály?

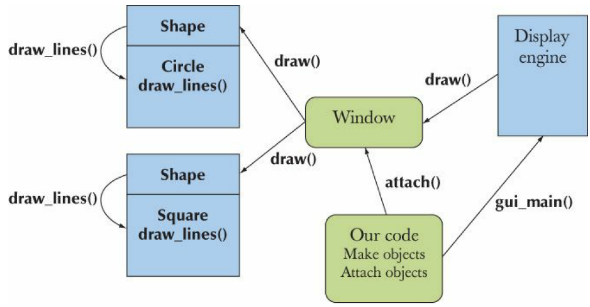
A Shape konstruktorai protected-ek, ami azt jelenti, hogy a Shape-ből eredeztett osztályok használhatják csak. Tehát nem tudunk csak úgy csinálni egy Shape-t, nincs olyan hogy Shape. Csak Shape-ből készített dolgok vannak. Tehát akkor abszrakt egy osztály, ha csak mint ősosztály használjuk.

Megjeleneik a majd később használt virtual destructor, ami minden Shape esetén felszabadítja az esetlegesen lefoglalt memóriát, hogy ne legyen memory leak

Vannak virtuális függvények, amiket alosztályok felülírhatnak, mint a draw\_lines(), ami máshogy működik egy téglalapnál, mint egy Text-nél.

Illetve látszik, hogy a szín, vonalstílus, pontok, privát tagok, tehát csak az arra megírt elérő függvényekkel tudjuk őket változtatni.

**Na de hogyan rajzol a Shape?**

A fő függvényünk a draw() függvény, mely beállítja a megfelelő stílust, színt, majd meghívja a draw\_lines() függvényt, ezzel biztosítva, hogy minden objektum úgy legyen megrajzolva, ahogy kell. A draw nem kezeli, hogy látható-e az objektum, vagy mi a kitöltőszín, ezt már a draw\_lines() kezeli. Tehát a Shape, maga nem nagyon rajzol ki semmit, csak beállítja az alapszíneket, majd odaadja a különböző típusú objektumoknak, hogy a saját megfelelő módjukon rajzolják meg amit kell. Ezt jelenti a virtual draw\_lines() a Shape-ben, hogy majd minden alosztály felülírja a sajátjával a Shape draw\_lines()-át, ha kell.   
Tehát, ha egy alosztálynak van egy ugyanolyan nevű és típusú függvénye, mint a virtual függvény az ősosztályban, az felülírja az ősosztályban lévőt, ez az overriding. Az ablak hívja a Shape draw()-ját, ami hívja a draw\_lines()-t, amit felülírhat az adott objektum a sajátjával.  
A move() függvény ugyanígy virtual, és az alosztályok felülírhatják.

**Másolás**

Röviden: Nincs.   
Copy konstruktor, és a copy assignment is törölve van, mivel egy Kör-t, ami egy Shape, nem tudunk bele másolni, egy Rectangle Shape-be.   
Szóval ja, másolást felejstük is el. Ha másolni akarunk, lehet arra írni egy function 🡪Clone()

**Ős és alosztályok**

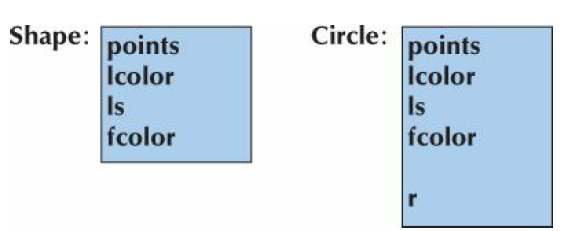
Deriválás (nem matek nyugi, nem kell a ptsd) 🡪 Eredeztetés magyarul, azaz A kör, egy fajta Shape, azaz a kör megörökli az összes tagját a Shape-nek.   
Derivált osztály=Alosztály , Ősosztály=SuperClass

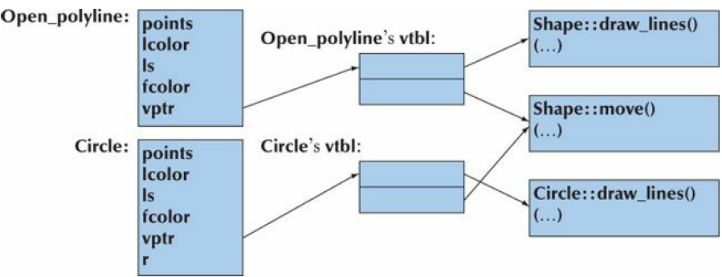
Virtuális függvényekről már beszéltünk, az ősosztály egy függvénye, melyet felülírhat az alosztályé. Ez futási idejű polimorphizmus (Run Time Polymorphism).

A Private, és Protected tagok segítségével meg tudtuk valósítani, hogy csak az férjen hozzá kulcs információhoz, akinek kell, lásd alosztályok.

Itt a teljes Hierarchia.  
A Shape-ből ered minden, de tovább is eredhetnek osztályokból más osztályok.

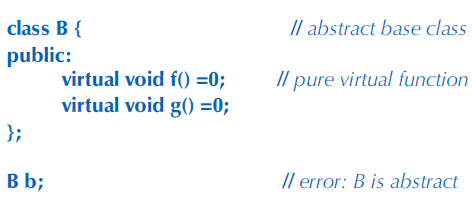
**Memóriában való elhelyezkedés**

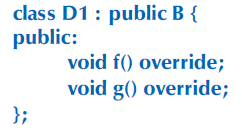
Szép is jó, amikről most beszéltünk, de mivel csak névben magasszintű programozás, ezért a memóriakezelés egy elég fontos pont.  
Minden alosztálya a Shape-nek, megörökli az összes tagját a Shape-nek, majd „hozzáteszi” a sajátjait.

Ahhoz hogy virtual függvényeket tudjunk kezelni, kell egy ún. Vtbl, (virtual function table) . Ez alapján hívja meg a megfelelő függvényt a compiler.   
Tehát az Open Polyline-nak nincs saját draw\_lines(), se saját move() függvénye, tehát mindkettőnél a Shape ezen függvényei hívodnak meg, míg a Kör-nek saját move()-ja szintén nincs, viszont van saját draw\_lines() függvénye, tehát azt hívja meg a Shape draw\_lines() függvénye helyett. Az Open Polyline-nak nincsen semmilyen saját függvénye, tehát lényegében egy sima Shape, tehát a vtbl csak a Shape függvényeit tudja/fogja hívni.   
Majd ezután a könyv elkezd beszélni arról, hogy OOP-hoz kell-e nekünk egyáltalán ez a memória dolog. A válasz=nem. Lmao

**Overriding, és Pure Virtual Functions**

Szóval, ha akarunk Virtual függvényeket használni, és felülírni őket akkor megírjuk az alap virtual függvényt elsősorban. Ezután, ha akarunk egy függvényt írni, ami ezt felülírja, pontosan ugyanolyan nevű, típusú kell hogy legyen az felülíró függvény, és a virtual függvény osztályából kell eredjen.

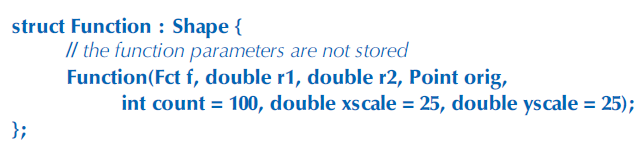
**Pure Virtual Functions - PVF**

Hogy mit, és hogyan?  
Szóval úgy készítünk PVF-et, hogy virtual void f()=0  
Ezzel azt csináljük, hogy ezt a függvényt egy alosztálynak felül KELL írnia.   
Hogy ez mire jó?  
Ezzel is abszrakttá tehetünk egy osztályt, amellett hogy néhány tagját protected-dé tesszük.  
A B-ből eredő összes osztály is absztrakt, amíg nem ír felül MINDEN PVF-et. Ha felülírja mindet, akkor már nem absztrakt, és példányosítható.

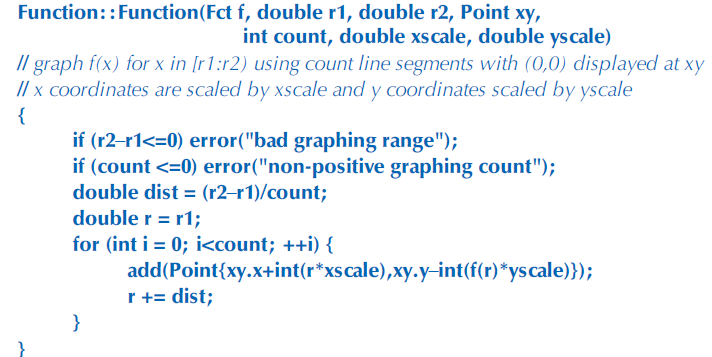
**Hogy miért is jó az OOP?**

Öröklés ijen=lyó  
De amúgy tényleg tud az öröklés sok mindent egyszerűsíteni.  
Emellett hozzá tudunk adni új típusú Shape-eket, anélkül hogy a meglévő kódot módosítani kellene. Ezzel a fenntartást sokkal egyszerűbbé téve.

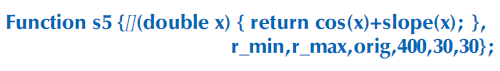
De ja idk

**Chapter 15. – Adatok és Függvények ábrázolása**

**Függvények**

Szóval, tudunk csinálni függvény objektumokat, amiket szintén tudunk ábrázolni  
A Függvény is egy fajta Shape, ami ki tud rajzolni egy függvényt. Itt nem sok elméleti dolog van, max az, hogy vannak az értékeknek amiket meg lehet adni default állapotaik, és még lehet adni kb mindent is.

**Lambda Kifejezések**

Mi az a Lambda kifejezés?  
Röviden egy névtelen függvény. Jelen esetben, amikor ki akarunk rajzolni egy függvényt, akkor az egyik argumentum, az maga a kirajzolandó függvénynek a megadása. Írhatunk erre egy külön függvényt, de mivel máshol nem fogjuk használni azt a függvényt, így elég feleslegesnek tűnik.  
Ezért használhatunk Lambda kifejezéseket.

Ahelyett, hogy megírtunk volna egy függvényt, ami cos(x)+slope(x) lenne, csak raktunk egy []-t, és utána írtuk meg ezt a lényegében „egyszerhasználatos” függvényt.

**Axis**

Egy csodás, Shape-ből eredeztetett objektum, ami lényegében nem más mint két vonal, és azokon a vonalakon jelzések. A kód tök érdekes meg minden de engem annyira nem érdekel tbh.  
Van saját draw\_lines() függvénye, saját set\_color() függvénye, saját move() függvénye.

**Approximation**

Van egy rész arról, mennyire is pontosak a számítások, amiket odadobunk a c++-nak.

**Clusterfuck**

A fejezet további részét őszintén nem értem. Vagy legalábbis azt mi volt az elképzelés  
Van egy rész, hogy hogyan olvasunk be dolgokat, nem mintha ne vettük volna ezt elméletileg bev-progon, meg van rá két egész chapter lmao (10,11)

Aztán arról beszél, hogy ne legyen ronda a grafikus felület, majd elkezd a skálázásról beszélni.

Végül, a bekért adatokból, hogy csinálunk grafikonokat.

Mi  
A  
Fasz

**Chapter 16. – GUI-k**

**GUI lehetőségek**

Konzolos interakció 🡪 Technikai, szakmai dolgokhoz jó lehet

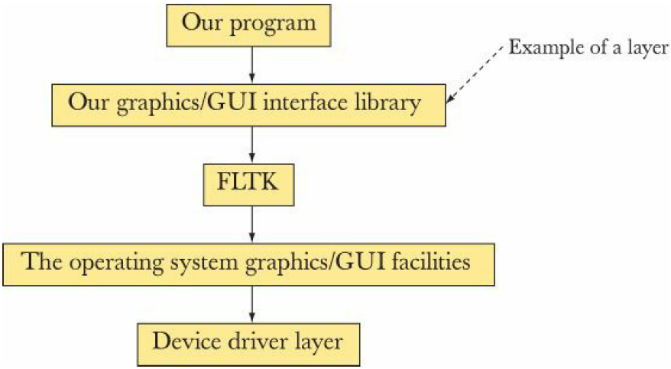
Grafikus Felhasználói Felület 🡪 Vizuális dolgokra (Duh), és amikor az interakció a képernyőn lévő dolgok változtatásáról szól

Böngésző 🡪 Jó tud lenni

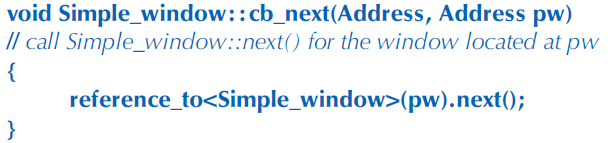
**Mi az a Wait\_for\_button()?**

A Wait\_for\_buttin() hasonlít egy kicsit input kéréshez. Amikor cin>>Anyád bekérés történik, akkor várunk a felhasználóra, hogy megtörténjen az input az Anyád nevű változóba.  
A wait\_for\_button esetén ezzel szemben arra várunk, hogy történjen valami, a next gomb megnyomása, vagy valami felhasználói interakció, amit viszont ellenőrizni kell.Ezt a GUI tesz meg. Ha valamit akarunk egy grafikai felületen, akkor először el kell mondani mire figyeljen a GUI, hol keressen kattintást, ha megtörténik mit csináljon a GUI, és újra várni amíg történik valami.

**Callback Függvény**

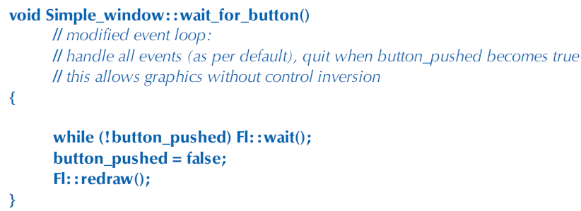
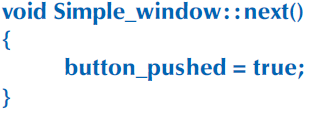
Mit jelent az, hogy Callback függvény? Ezt hívja meg a GUI, amikor valamilyen eseményt érzékel, mondjuk kattintás.   
De miért is kell a Callback függvény?   
Először nézzük meg mi is történik, amikor kattintunk egyet.   
Szóval a mi programunk több rétegen van, előszöris alatta van a GUI Library, az FLTK, az FLTK pedig az Operációs rendszer grafikai rendszerét használja, ami pedig az eszköz illesztőprogramját. Ha egy kattintásról van szó, az egér illesztőprogramjának érzékelnie kell ezt a kattintást, és valahogy meg kell hívnia a callback függvényt. A callback függvényünk és az ablakunk memóriacímét a rétegeken keresztül „leküldjük”, és valami „kód lent” meghívja ezt a callback függvényt amikor érzékeli a kattintást. Mivel a GUI, és az Oprendszer nem csak C++ kóddal foglalkozik, és nem tudja mi az ablak, vagy egy gomb – sőt nem tud semmilyen osztályról sem amit írtunk – mily meglepő – ezért a callback függvényt univerzális módon kell megírni, mivel egy nagyon alacsony szintű dologról beszélünk. Egy callback függvénynek nincs visszatérési értéke, és két memóriacímet kér argumentumként.

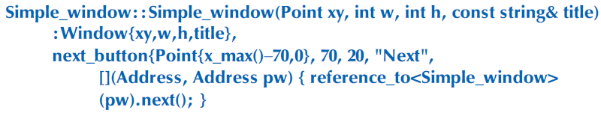
A Static azért kell, hogy egy általános függvényként legyen meghívva, ne mint c++ tagfüggvény.  
Itt nagyon közel vagyunk a hardwerhez, ezért nincsenek meg a támaszaink a c++-tól/nyelvtől. Az első argumentum a GUI objektum, a widget, amire hívodott a callback, ez nekünk jelen esetben nem kell, a második viszont az ablak címe, ami tartalmazza az adott widgetet.

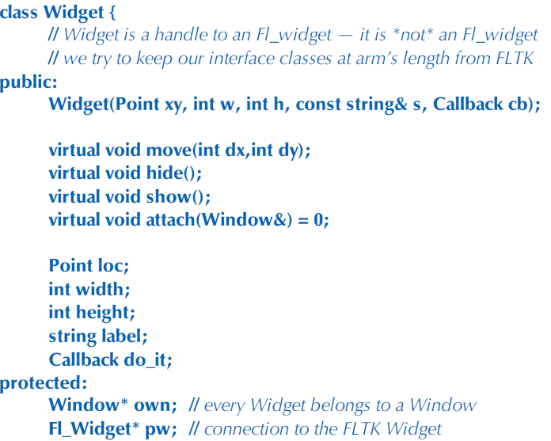
Itt a reference to azt mondja meg a compilernek, hogy a pw az ablakunk memóriacíme. Így már tudjuk használni a next gombunkat, jelen esetben.

Megírhattuk volna az egészet a callback függvényünkön, a cb\_next()-en belül, de az átláthatóság, és a könnyű kezelhetőség miatt döntöttünk úgy, hogy lesz egy külön next() függvény, és egy cb\_next callback függvény.

**Oké, megírtuk a next callback függvényét, de mire használjuk?**

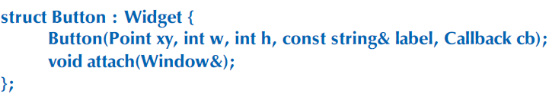
Szóval mostmár tudunk érzékelni gombnyomást, great!   
De mi a faszra megyünk vele?  
Itt jön be a wait\_for\_button().   
Alapvetően a wait\_for\_button() arra jó, hogy kirajzolja az eddigi változásokat. Amíg a gomb „nincs megnyomva”, addig vár, majd amint megnyomásra kerül a gomb – amit a callback függvényünk érzékel – újrarajzolja az ablakon lévő dolgokat, majd újra vár, a következő gombnyomásig. Ezt az FLTK Wait() függvénye teszi lehetővé.

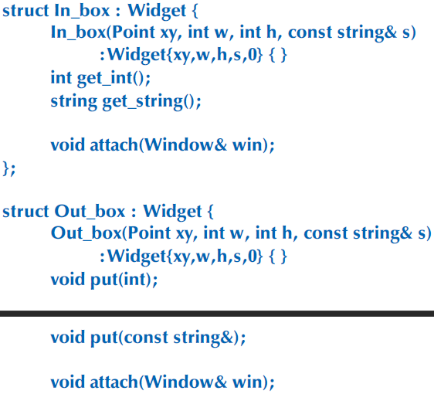
Ha igazán átláthatatlanná akarjuk tenni a dolgokat, vagy a drillt – ahogy én csináltam – callback függvények írása helyett használhatunk minden callback függvény helyett egy lambda kifejezést, mivel másra nem nagyon használjuk ezt a függvényt.

**A Widget ősosztály**

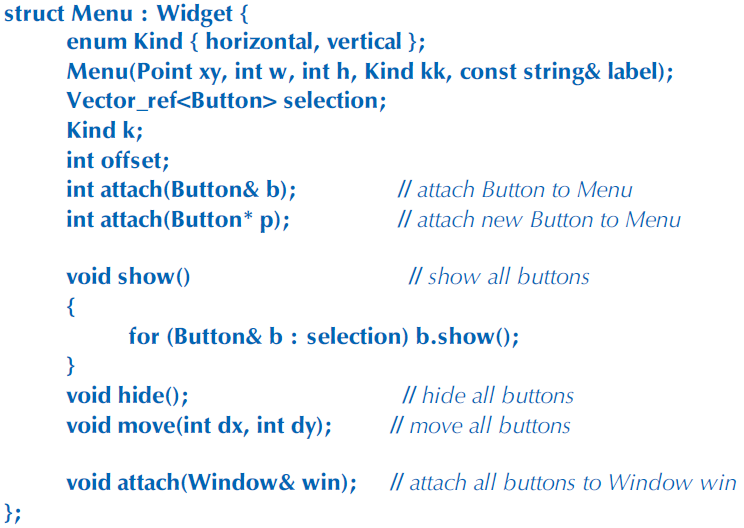
A Widget, valami olyan dolog, mellyel interaktálni tudunk a felhasználóval a GUI felületén keresztül. Az implementációja igazán csodás, van egy csomó virtuális függvénye, amit a widget objektumok majd felülírnak. Emellett minden widgetnek megvan a bal felső pontja, szélessége, magassága, a felirata, és egy callback függvénye, hisz ezekkel akarunk interaktálni a felhasználóval.

**A gomb**

A gomb egy widget, mert a felhasználó meg tudja nyomni, ezzel interaktálva a programmal. Mivel a gombnak nincsen saját függvénye, csak az attach – de az Pure Virtual Function a widgetben, tehát lennie kell valaminek, ami felülírja – ezért megörökli a widget függvényeit. A mozgatást, illetve a hide() és show() függvényeket, melyekkel láthatatlanná, illetve láthatóvá lehet őket tenni.

**In\_Box / Out\_Box**

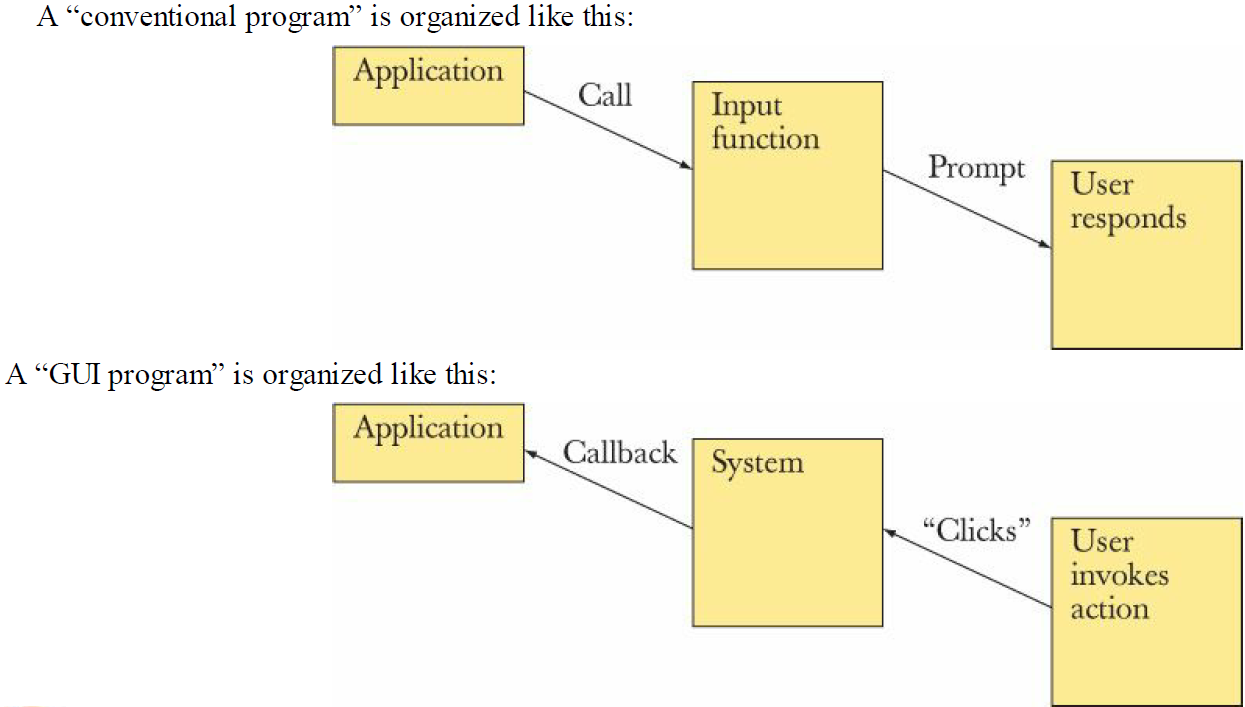
Két újabb widget, melyek elég hasznosak. Az In\_Box egy, „doboz” amire a felhasználó írhat dolgokat, tehát lényegében egy grafikus input felület adatok bekérésére. Az Out\_Box pedig egy output felület, ahova ki tudunk írni eredményeket, adatok a felhasználó számára. Van két függvényük, int-ek és string-ek inputjára, illetve outputjára. Itt csak egész számokat kezelünk, de lehetne bármi is, csak nem vagyunk olyan mazochisták, hogy komplex számokat kéregessünk be lmao.

**Menu**

A menü leegyszerűsítve egy vektor gombok tárolására. Szokásosan van egy pont, ami a bal felső sarka, szélesség, magasság, a kind azt jelenti, hogy vízszintes, vagy függőleges menü, illetve van egy címe. A menühöz hozzá tudunk csatolni gombokat, legpraktikusabb   
new segítségével névtelen gombokat.

A 16. drill feladata szépen bemutatja a menü működését, de ez egy elméleti összefoglaló, szóval azzal nem fogok foglalkozni.

**Control Inversion**

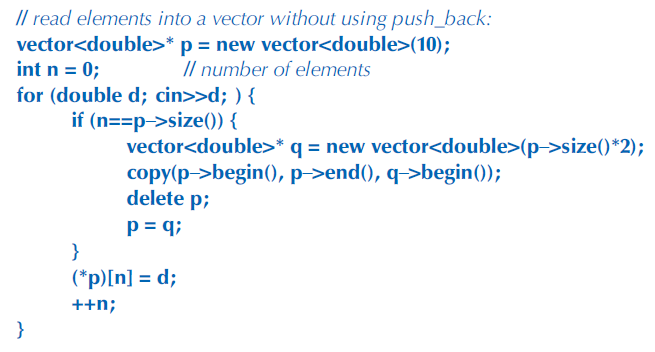
Mivel grafikus felületekkel dolgozunk, megváltozott a dolgok menete egy „konvencionális” programmal szemben. Egy átlagos programban a program meghív egy input függvényt, amire a felhasználó válaszol.  
Egy grafikus program azonban máshogy néz ki. A Felhasználóval kezdünk, aki csinál valamit,  
 - kattint például – ezt a rendszer érzékeli, a rendszer pedig a callback függvény segítségével eljut a programhoz, ami megcsinálja, amit kell a callback alapján. Tehát minden a felhasználótól függ, hiszen tőle indul minden folyamat. Ez erősen megnehezíti a tesztelést, debugolást.

**Chapter 19. – Vektor, Templatek, Kivételek**

**Bevezetés**

Szóval a Chapter 17-18-ban megírtunk egy saját Vektor osztályt, amiben tudtunk tárolni double-t, tudtunk másolni, és nem volt benne memory leak. Elvileg(definitely notforeshadowing).   
Emellett hozzá tudtunk férni a vektorunk elemeihez úgy, ahogy a standard library vektorja.

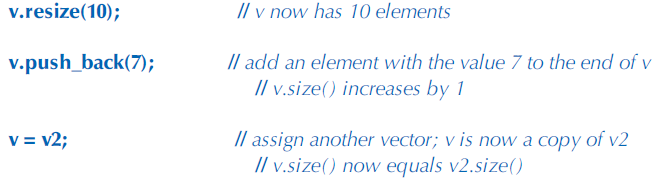
**Problémák:**

Hogy változtatjuk meg a vektor méretét? Hogy kezeljük az out-of-range errorokat? Hogyan adjuk meg milyen típusú változót akarunk benne tárolni? Hogy írunk egy olyan vektort, amiben tetszőleges típusú változót tudunk tárolni?  
Két fő rugalmasságot szeretnénk. A mérete ne legyen adott, lehessen változtatni, illetve típusfüggetlen legyen. Többféle tároló van, a vektor talán a legszélesebb körben használt, ezért ezeken a területeken kihagyhatatlan az efféle rugalmasság.

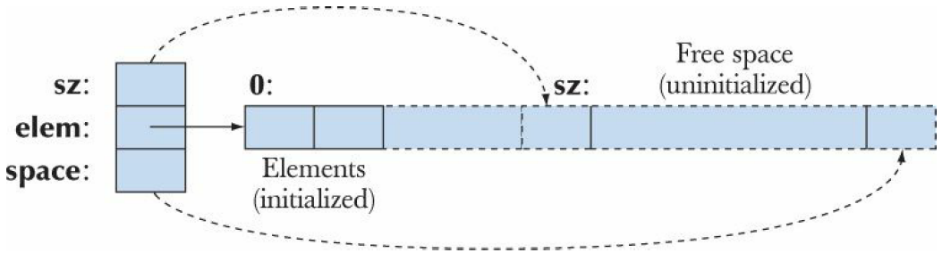
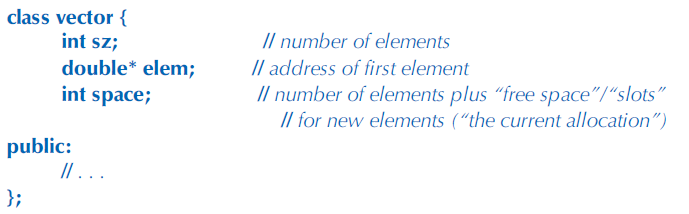
Legtöbbször a push\_back-et használjuk, és ha akarunk rugalmas méretet, itt kéne ellenőrizni túlnőttünk-e a méreten. Meg lehetne enélkül is csinálni, ahogy a példa mutatja, azonban itt elég alacsony szintű dolgokat hozunk olyan helyre, ahol könnyű hibázni, we no likey.  
Az is egy hozzáállás, hogy csak adjunk alapból elég helyet a vektornak ’oszt kész. De mennyi az elég hely?   
Van-e olyan, hogy elég hely? Miután egy photoshop projectem egyszer csinált egy 90 Gb-os TEMPORARY filet, én amellett vagyok, nincs.

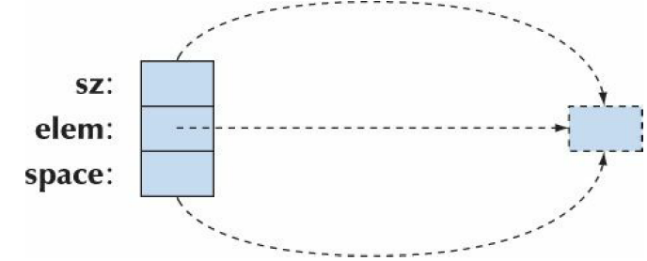
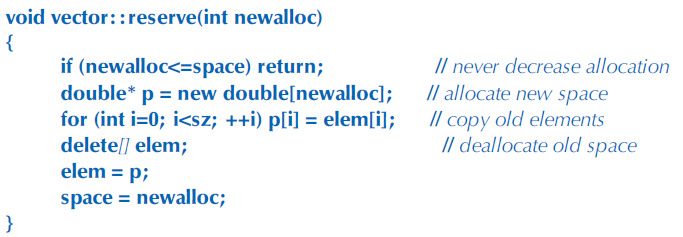
Sokféle tárolóról beszélhetünk, és lehet, hogy vannak tárolók melyek alkalmasabbak néhány dolog tárolására, mint egy vektor, de nekünk itt az a célunk, hogy minél rugalmasabb legyen az általunk írt Vektor osztály.

**Méretváltoztatás – As a lot of people would like**

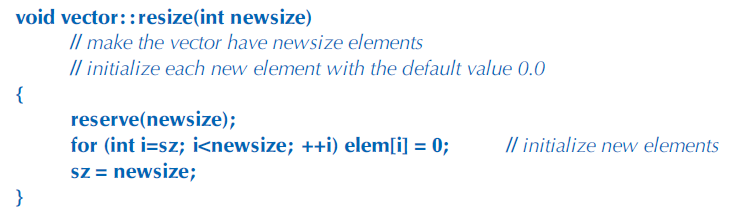
Mivel a Standard Library vektorját másoljuk lényegében, hogyan oldja meg a std\_lib?  
Így:

Na ez sokat nem segít az az igazság, mert nem igazán látjuk, hogy ezek mit is csinálnak

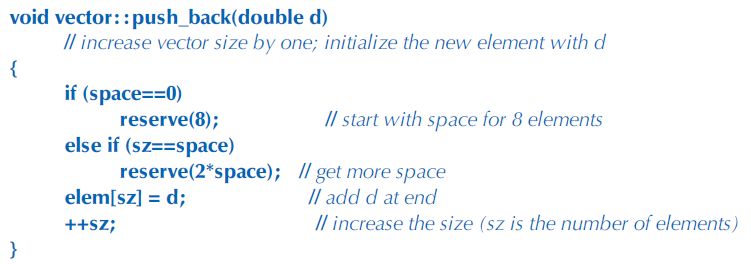
Így néz ki a vektorunk jelenleg, és a memóriabeli vizualizáció. Mivel 0-tól számolunk, az sz lényegében az utolsó elem utánra mutat, a space meg arra, hogy mennyi hely van jelenleg lefoglalva.

Amikor egy vektort létrehozunk alapvetően így néz ki. Mivel az alap konstruktor az elemszámot 0-ra teszi, az első elem egy Nullptr, és a rendelkezésre álló hely szintén 0.   
**Reserve – Az alapkő**

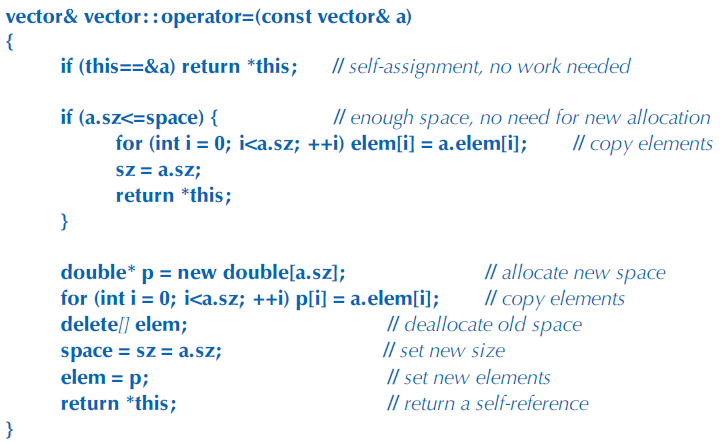
Méretváltoztatásnál a reserve az amire támaszkodunk, lényegében ez csinál mindent. Először lefoglal x memóriaterületet, ami nagyobb, mint az eddigi, majd odamásolja az elemeit a vektornak, viszont az új memóriaterületen nem inicializál semmit, az majd a push\_back(), és a resize() feladata. Ezután a memóriaterületet, ahonnan a “régi” elemeket átmásolta az új területre felszabadítja, majd beállítja az első elemét az új helyen, és hogy mennyi a rendelkezésre álló hely.

**Resize**

Így, hogy megvan a reserve, így már elég egyszerű a resize() implementálása. Itt már nem csinálunk semmi érdemlegeset, lényegében mindent a reserve végez. Ha csökkenteni szeretnénk a területet, akkor a reserve dobja vissza, ha megegyezik szintén.

**Push\_back**

Szintén, a reserve végzi a munka nagyrészét, és elég könnyű dolgunk van. Ha a space=0, azaz a vektor üres, - valamiért random - 8 helyet adunk neki.  
Ha elérte a limitet, és több hely kell, akkor megduplázzuk a rendelkezésre álló helyet. Ezután elvégezzük a push\_back tényleges dolgát, és hozzáfűzzük a vektor végéhez az adott elemet, az elemszámot 1-gyel növeljük.

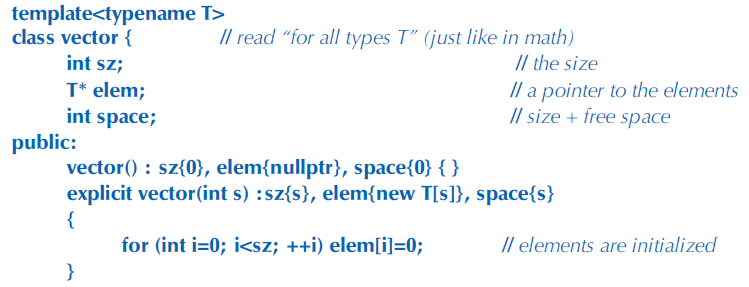
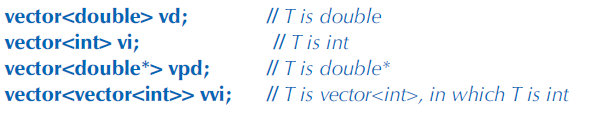
**Copy Assignment**

Megkönnyíthettük volna az életünket azzal, hogy ha azt mondjuk, hogy csak olyan vektorokat lehet „egymásba” másolni, amiknek a mérete megegyezik, but we don’t do that here  
Az az alap elképzelés, hogy ha v1=v2, azaz v1, v2 egy másolata. Három lehetőség van. Ha egy vektort önmagába másolunk, akkor nincs dolgunk.   
Ha v1-ben több a hely, mint v2-ben, akkor csak átmásoljuk az elemeket.  
Ha kevesebb a hely, akkor allokálunk helyet, átmásoljuk az elemeket az új helyre, és a régi helyet deallokáljuk, beállítjuk a rendelkezésre álló helyet, elemszámot, és egy self referenciával térünk vissza konvenció szerint.

**Típusfüggetlenség**

Eddig a méretről beszélgettünk – többé kevésbé – de most beszéljünk a másik fő pontról, hogy többféle változóval is működjön a vector. Most kezdődik a generikus programozás, illetve annak az alapja.

**Templatek**

A templatek-nek az lényege, hogy egy típust helyettesít.   
Itt ez a csoda. Ezentúl minden helyre, ahova eddig a típust írtuk, azaz double-t, mostantól T-t írunk, ezzel megoldva, hogy T-nyi helyet foglal le a memóriában, attól függően, hogy egy T mennyit foglal.

Lényegében a compiler a T helyére behelyettesíti azt a valtozótípust, amit megadtunk, a program egészébe, minden egyes helyre, ahol T van.

Ezeken a generikus vektorokon ugyanúgy tudjuk használni a tagfüggvényeinket, a push\_back-et, a reserve-t, hiszen a reserve majd T-nyi helyet foglal le a memóriában.

A templatek a generikus programozás alapjai, ezzel tudunk lényegében mindent általánosítani, univerzálissá tenni. Az OOP és a Generikus programozás közötti legnagyobb különbség az, hogy milyen függvény fut le, az Generikus programozás esetében Compiler dönti el, azaz compile time, míg OOP esetén Run-Time-kor „derül ki”.

**Generikus Programozás vs OOP**

Mivel Stroustop bácsi szereti az OOP-t, elkezd beszélni mindkettő típusú programozás előnyeiről, hátrányairől, mintha kicsit az OOP felé húzva.

Szóval a Generikus programozás alapja a templatek, compile idejű. A rugalmasság, és a gyorsaság ára az, hogy nehezen elkülöníthető a template belseje, és az interface-e. Emiatt nehéz debugolni, és érdekes error üzeneteket adhat. A Template definíciók headerekben vannak, mivel a mostani compilerek-nek a template teljessége kell, hogy definiálva legyen. Az összes tagfüggvény, és template függvény. Amíg rövid, egyszerű dolgokhoz használunk templateket, addig nincs baj, de nagy rendszereknél könnyen bele lehet bonyolódni.

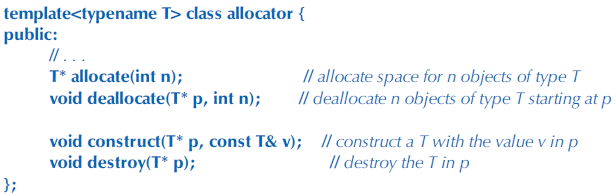
Ezzel szemben nem olyan gyors az OOP, de elvileg könnyebben fenntartható, átlátható. Idk tbh

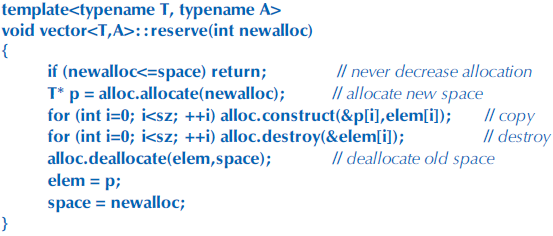
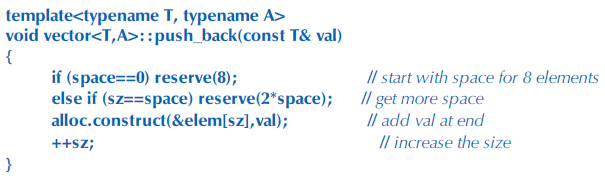
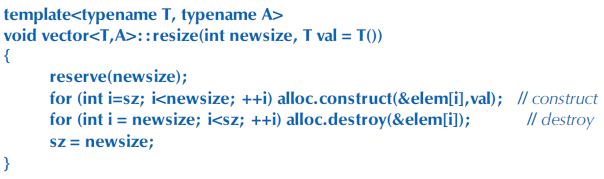
Lehet-e keverni az OOP-t, és a Generikus Programozást? TLDR: Nem.

**A vektorunk generalizálása**

Szép és jó, hogy egy template-et használunk a vektorunkhoz, de ezzel miben változik az implementáció?

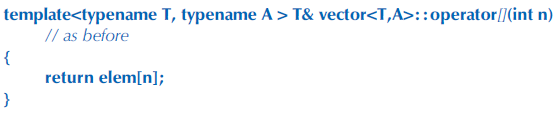
Például, mi legyen az alap érték? Egy int-nél másnak kell lennie, mint egy string-nél. Egyszerű megoldás lenne, ha nem adnak default értékeket, de hát az túl egyszerű, meg hát, ha már generalizálunk akkor ne legyen ilyenek. Ok then  
Erre a megoldás, hogy a legtöbb – alap legalábbis – típusnak van egy default értéke, és beállíthatjuk, hogy a T elemű vektor elemeinek alap értéke legyen T alapértéke.   
De lehet, hogy T-nek nincsen alapértéke.

Másik nagy kérdés, hogy szabadítjuk fel a memóriát?  
Azért egy nehéz kérdés, mert mivel nem biztos, hogy T-nek van alapértéke, ezáltal nem biztos, hogy minden adatunk inicializált.   
Eddig megpróbáltuk teljes mértékben elkerülni az inicializálatlan adatokat, mert érdekes errort adhatnak. (elvileg) Tehát kell valami, amivel tudjuk kezelni az inicializálatlan memóriát. Erre a megoldást a standard libraryben található allocator adja. Lehetőséget ad hely allokálására, deallokálására, tudunk készíteni egy T objektumot az iniciálizatlan helyen, és törölni is tudjuk.  
(itt kicsit a fogalmak angolról magyarra fordítása nehezen ment, ez nem biztos hogy pontos)

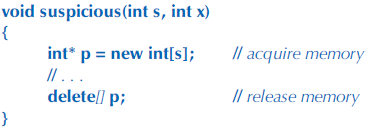
Az allocator implementálásával a reserve kell, hogy változzon legfőképp, hiszen az mindennek az alapja, de a resize és a push\_back is minimálisan változik.

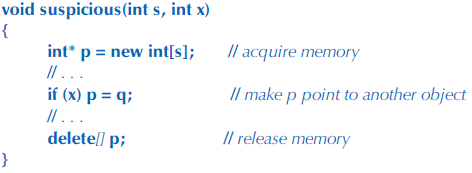
**Range Checking, és kivételek**

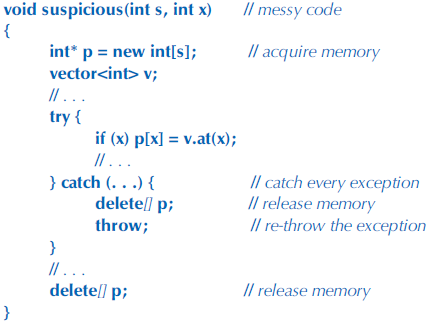
Eddig a vektorunk nagyon szép, és jó, de még nem kezeltük le, hogy használatakor nem próbálnak-e lekérni egy nagyobb indexű elemet, mint ami a maximum. Ennél jobban nem tudom leírni, de értitek na.

Szóval csinálunk egy csodálatos at() függvényt, ami megmondja, nem kértünk-e túl nagy, vagy túl kicsi elemet.   
Egészen csodás.  
Emellett megcsináljuk a subscript operátort is, hogy könnyen elérjük az elemeket, ellenőrzés nélkül.   
De miért nem ellenőrizzük a Subscript operátorban? Illetve miért nem lenne elég csak ott ellenőrizni? Hát Stroustop bácsi nagyon szépen elmagyarázza, hogy hát kompatibilitás, régebb óta nem ellenőrzötten használják, mint hogy lenne c++ hibakezelés. Emellett a nem ellenőrzött hatékonyabb is, meg elvileg vannak olyan helyzetek ahol a kivételek elfogadhatalanok. If you say so boss

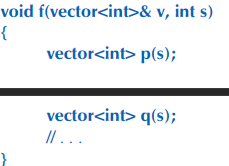
Amit eddig használtunk Vektor egy olyan verzió, amiben van ellenőrzés Subscriptnél is, de ez nem adott. A standard library egy macrot használ, hogy a vector legyen egyelőre a Vectorral, és a nagy V betűvel kezdődő vektorban le van kezelve az out of range error.

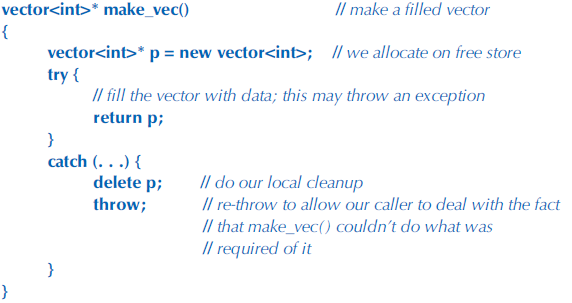
**Memóriakezelés újabb csodái**

Szóval itt ez a program ,szép és jól működik. Nagyon egyszerű, kérünk memóriát, csinálunk valamit, majd felszabadítjuk dolog. Its kinda sus tho

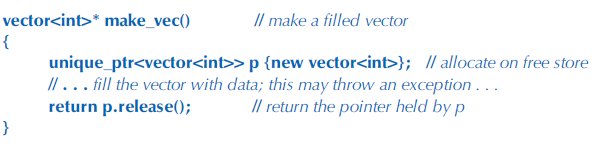
Viszont ezzel lehet probléma. Ha mondjuk átállítjuk p-t, hogy máshova mutasson, akkor felszabadítjuk azt a helyet, ahova a p adott pillanatban mutat, de a régi hely az nem szabadul fel. Jelen esetben nem tudjuk, hogy mi az eredménye az if-nek, lehet, hogy átállítjuk a p-t, lehet nem. Az is lehet, hogy nem szabadítjuk fel, mert valami kivételt dobunk, még a felszabadítás előtt.

Erre vannak csodás sufnimegoldások, mint egy try-catch, de ezzel az a probléma, amikor kétszer próbálunk meg felszabadítani egy memóriaterületet.

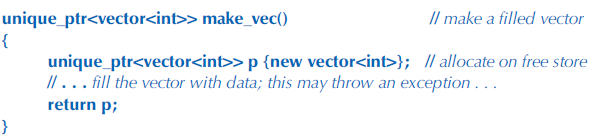
Erre egy egyszerű megoldás az f, mivel ezzel kikerüljük, hogy bármit is kelljen csinálni memóriával, majd a vektor destruktora elintézi nekünk, amit kell.   
Ezért is tárolunk adatokat vektorban mondjuk, és nem a free store-ban közvetlen, sokkal egyszerűbbé teszi a dolgunkat.

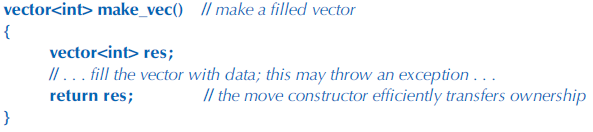
Na de mit csinálunk, amikor a vektort nem csak egy scope-ban használjuk?

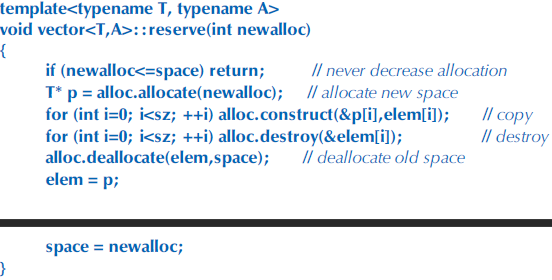
Ezt:   
Egy elég gyakori módja az errorkezelésnek.   
Megpróbál valamit csinálni a függvény, ha nem sikerült neki akkor eltakarítja maga után amit kell, ha sikerült neki akkor meg happy mindenki. Jelen esetben egy vektort készít, amihez foglal le memóriát. Ha eközben hiba van, akkor hibát is dob, majd ahogy írtam feltakarít maga után a függvény.   
Ez egy egyszerű, és hatékony módja az ilyen memóriakezeléssel kapcsolatos hibák kezelésére.  
Ezt hívjuk alap garanciának. Van az erős garancia, ami ugyanezt csinálja, csak emellett biztosítja, hogy ha hibát dob a függvény, akkor minden érték a függvény meghívása előtti érték legyen. Van valami no-throw garancia, de azt nem nagyon értem tbh.

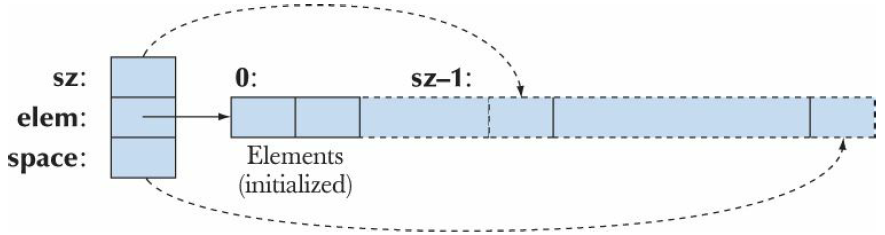
Szép és jó ez az alap garancia, amihez példát is néztünk, de elég ronda a kód. Erre a megoldás a RAII 🡪 Resource Allocation Is Initialization.   
Magyarul, kell valami objektum, ami eltárolja a vektorunkat, hogy error esetén fel tudja szabadítani a hozzá tartozó memóriát. Erre a megoldás a unique\_ptr.

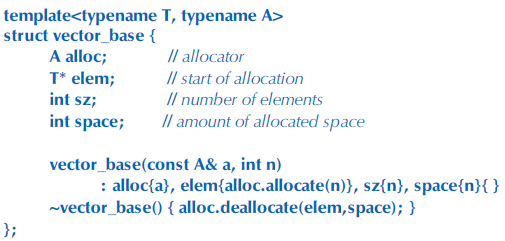
A unique\_ptr egy objektum, amiben egy pointer van. Egyből inicializáljuk a pointerrel, amit a new-tól kaptunk, amikor a vektort létrehoztuk. Szóval, igazából a unique\_ptr, ahogy a neve is mutatja, egy pointer.   
De mivel a „unique pointeré” a vektorra mutató pointer, amikor a unique\_ptr törlődik, kitörli az a vektorra mutató pointert is, azaz felszabadítja a vektornak szánt memóriát.   
Tehát, ha valami error van, amikor töltjük fel a vektort, vagy túl hamar térünk vissza a make\_vec-ben, akkor a unique\_ptr gondoskodik róla, hogy a vektor memóriája fel legyen szabadítva. A p.release() visszanyeri a pointert, ami a vektorra mutat, és p pointert „lecseréli” nullptr-re. Ez azért jó, mert így nincs meg a dupla memóriafelszabadítás problémája, hisz mit szabadítanánk fel egy nullpointeren.

Szóval ez az a verzió, ami teljesen megoldja a memory leak problémáját. Egy „hátránya” van, ahogy a neve is mondja, egy unique pointer, tehát nem lehet két unqique pointer, ami ugyanoda mutat.

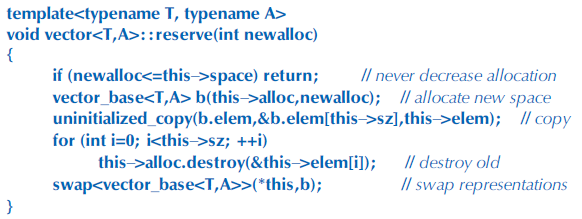
Az előző megoldás is tökéletes, azonban elég közel van a hardwerhez, ezért mivel hozzáadtuk a vektorhoz a „move” operátort, ezért megoldottuk már akkor a problémát. Csak használjuk a move konstruktort, hogy az foglalkozzon a memóriával.

Most, hogy már értjük (lmao) a memóriakezelés problémáit, hogyha ránézünk az addig jónak gondolt reserve()-re, akkor problémát találunk.   
Na, látod? Nyugi én sem.  
De a könyv felhívja figyelmet arra, hogy ha mondjuk akkor történik egy hiba, amikor másoljuk át a régi elemeket az új helyre, akkor a régi elemek memóriaterülete nem lesz felszabadítva.   
Használhatnánk unique\_ptr-t, de ha már megbeszéltük, hogy az a legjobb memóriakezelés, amit nem mi csinálunk, tartsuk magunkat ehhez.

A memória a vektornak, egy erőforrás ha úgy nézzük, és hogy ha úgy gondolkozunk mint a RAII, miért ne csinálnánk egy osztályt, ami a vektor memóriakezelésével foglalkozik, hogy majd ő maga eltakarít maga után, ha kell? Ezt is tesszük. Az alábbi ugye a koncepció

Szóval kell egy osztály, amiben az itt látható dolgokat reprezentáljuk.

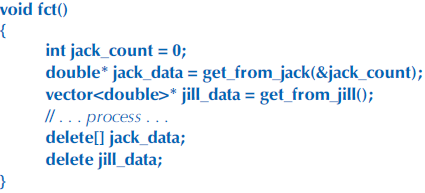
Íme itt ez a csoda, amivel csináltunk egy külön interface-t a memóriakezeléshez, hisz ezek eddig a vektorunkban magában voltak benne. Ahogy látjuk, vannak a vizualizált elemeink, a konstruktor, majd a destruktor, ami eltakarít mindent amit kell.

Ezután a reserve elég máshogy néz ki.

A memória allokálást a vector\_base-en keresztül végezzük, hogy ha bármilyen hiba lenne, majd a vector\_base destruktora elvégzi a felszabadítást.   
Használunk egy std\_lib függvényt, a copy-nál, mert ez a könyv szerint jobb mint egy ciklus. You do you fam.   
A swap függvény is egy std\_lib dolog, ami megcseréli két dolog értékét.

Na megvan ez a faszság is.

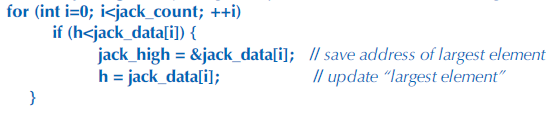
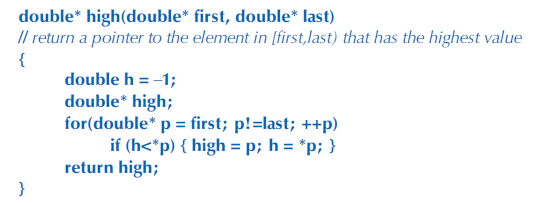
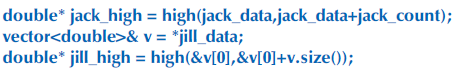
**Chapter 20. – Tárolók és Iterátorok**

Jön a nagy kérdés, hogy tároljunk adatokat hatékonyan?  
A könyv példája:   
Kettő ember sebesség adatokat mérnek egy programmal, egyikük tömb-ben tárolja el, másikuk egy vektorban. Ha mi használni szeretnénk ezeket, hogyan tudjuk ezt megtenni?  
Ha mindkettőjük programja mondjuk egy file-ba írja ki az adatokat, könnyű dolgunk van, csak a 10-11. fejezeti dolgok szerint dolgozunk velük.   
Ha viszont erre nincs lehetőség, vagy nem egy jó opció, nehezebb a dolgunk.   
Tegyük fel (mert a könyv ezt a példát hozza) hogy az adatok gyűjtése egy függvénnyel történik, amit másodpercenként meghívunk, hogy kapjuk az adatokat. A legnagyobb probléma az, hogy nem tudjuk, hogy ketten akiktől kapjuk az adatokat, hogyan tárolják azokat. Két lehetőségünk van-  
Abban a formában ahogy kapjuk, kell valamit kezdenünk vele, vagy beolvassuk és úgy tároljuk ahogyan mi szeretnénk.

A kérdés, mit akarunk az adatokkal? Rendezni? Átlag, max érték? Összehasonlítani a kettőt? Csinálhatunk sokféle dolgot, a könyv a max érték megtalálását hozza fel példának.

**Általánosítás**

Azt szeretnénk, hogy valami univerzális módon tudjuk kezelni mindkettő adatot, attól függetlenül, hogy az egyiket egy tömb-ben, a másikat pedig egy vektorban tárolják.

Jelenleg így férünk hozzá kettejük adatainak számához, illetve magukhoz az adatokhoz. Ha referenciát használnánk, azzal minimalizálni tudnánk a különbséget, és írni egy függvényt ami mindkettőre működik

Egészen csodálatos, de egy clusterfuck tbh.   
És emellett pedig, csak erre specifikus két dologra működik, listával, vagy mappal nem működne

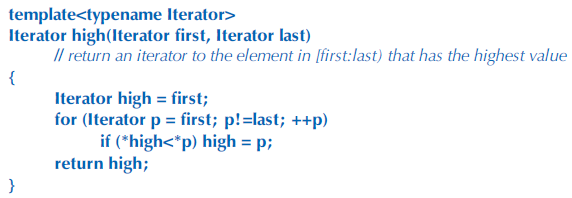
Szóval mit szeretnénk?  
Egy univerzális módot, amivel különböző tárolokkal tudunk dolgozni, kinyerni belőlük adatot, rendezni, stb. Erre lesz a megoldás az iterátorok használata.

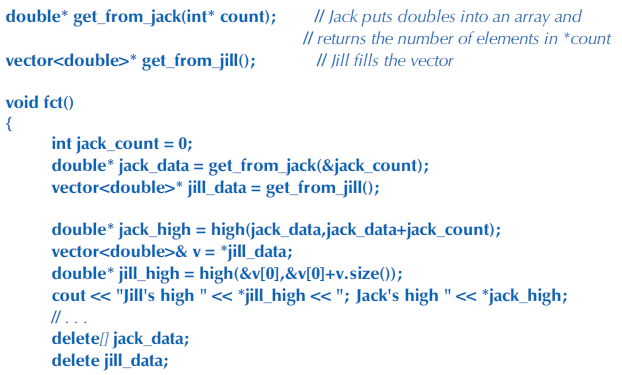
**Iterátorok**

A célunk az általánosítás. A legtöbb adatokon végzett művelethet, nem szükséges tudnunk, hogyan van tárolva maga az adat. Nem érdelenek minket a részletek, és végsős soron, minden tároló a memóriát használja, amit pedig mi is ismerünk és tudunk használni (lmao).

Szóval mik is az iterátorok?  
Először fel kell állítanunk egy koncepciót. Hogyan is tároljuk az adatokat?  
A könyv erre azt a választ adja, hogy mint egy sor. Egy sornak van eleje, és van vége.   
Tehát ha tudjuk, hogy hol kezdődik, és hol van vége, valamint tudunk a memóriában „lépkedni”, akkor bármit el tudunk végezni, mivel tudjuk honnan kell menni, hova.  
Egy iterátor lényegében csak egy pointer, ami egy „Különleges” helyre mutat. Ez a különleges hely lehet, amit előbb említettem.   
Van egy begin iterátorunk, ami egy adatstruktúra memóriabeli kezdetére mutat, illetve van egy end iterátorunk, ami a adatstruktúra memóriabeli végére mutat. Itt van egy jópár hasznos műveletünk.

* A==B 🡪 Két iterátor megegyezik-e
* A!=B🡪 Nem egyezik meg, if-ekben nagyon jól használható
* P\*, p\*=val🡪 Tudunk hivatkozni egy memóriacímre, ahogy eddig, is tudunk értéket is adni
* ++p 🡪 Az iterátorok léptetése

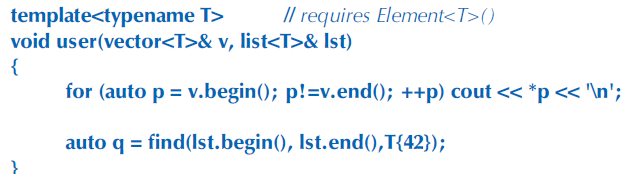
Ezek tudatában az előző algoritmust, mely a legnagyobb elemet keresi meg, elég könnyű általánosan megírni.

Van egy függvényünk, mely egy Iterátorral tér vissza, és két argumentumot kér, két iterátort, azaz memóriacímet. Kéri a tároló első, és utolsó elemét.   
Ezután könnyen tudunk egy ciklust írni, ami a tároló első elemétől indul, és amíg nem jut el az utolsó elemig, addig lépked előre, és ellenőrzi az adott elem nagyobb-e mint az eddigi legnagyobb elem, majd visszatér egy iterátorral, azaz memóriacímmel.   
Ennek az a szépsége, hogy lényegében az összes standard library-s tárolóból ki tudjuk nyerni az első, és utolsó elemet. Az első, az magára a tárolóra mutató pointer, az utolsót pedig úgy kapjuk meg, hogy az elsőhöz hozzáadjuk az elemek számát. (és abból lehet ki kell vonni 1-et)   
Ezáltal ez az algoritmus, ami megkeresi a legnagyobb elemet, bármilyen tárolóra használható, amiben számontartjuk az elemek számát.

**Láncolt listák**

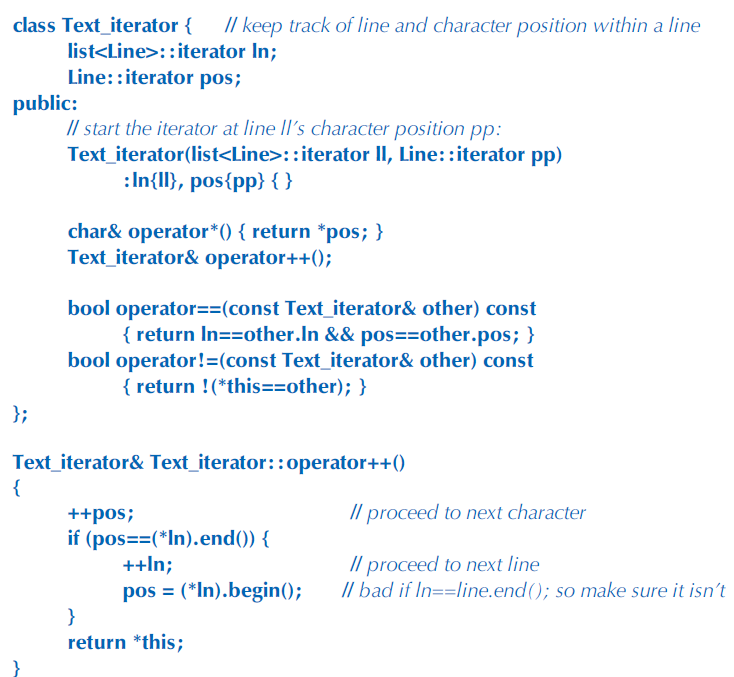
Igazából annak az implementációja, amit vettünk adatszerkből, illetve a duplán láncolt lista implementációja. Megmutatja ahogy van egy begin(), és egy end() függvénye a láncolt listáknak, ami az első, és utolsó elemet (technikailag utolsó utáni) adja vissza. Emellett bemutatja, hogyan implementálja az iterátorok léptetését, azaz meg kell írni a ++ -- == operátorokat, de ez szépen le van írva a könyvben, nem hinném hogy nagyon kell, az elméleti része meg bennevan az adatszerk jegyzetemben. Egy dolgot tudok még hozzáfűzni. Egy láncolt listának alap esetben három fő dolga van, az érték, egy prev mutató, és egy next mutató. A könyv kicsit más.

**Vektorunk további generalizálása**

Csodás minifejezet, ahol mivel egy univerzális vektorrol van szó, az iterátorokat, és azok műveleteit megírjuk, hogy minden típussal működjenek.

Illetve megismerkedünk az auto nevű dologgal, ami megkönnyíti az életünket, hogy ciklusok terén nem kelljen mindig megadni, hogy milyen típussal dolgozunk, hiszen a compiler tudja, hogy a p milyen típus, majd ő behelyettesíti magának.

**Példa**

Továbbá a könyv leír egy érdekes példát, hogy az iterátorokat mire lehet használni.   
Egy egyszerű szövegszerkesztőt hoz fel példának, ahol két iterátort hozunk létre. Van egy iterátorunk amiben azt tároljuk el hanyadik sorban járunk, illetve még egy iterátor, hogy azon a soron belül hanyadik karakternél.

**Vektorunk további írása – mert miért ne**

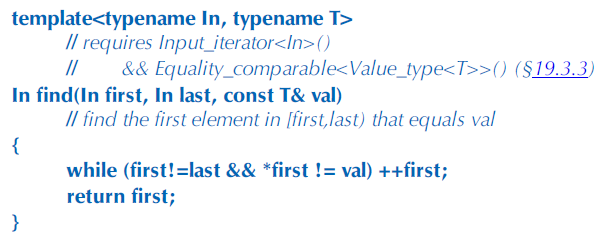
Alapvetően megvagyunk a legtöbb dologgal, és majdnem teljesen sikerült lemásolnia standard library-s vektort, de még nem vagyunk készen. Van még két műveletünk, az insert() és az erase(), amik egyébként sokkal jobban működnek egy listában, és elég erőforrásigényesek egy vektorban. Méghozzá azért, mert ha egy adott helyre akarunk beszúrni egy elemet, vagy kitörölni, akkor merülhetnek fel problémák, hiszen az iterátor ezt nem tudja, hogy onann kitöröltünk egy elemet, vagy hozzáadtunk egy elemet, szóval jobbra-balra kell másolgatni a vektor összes elemét, hogy konzisztensek maradjunk, és az iterátorok használhatóak legyenek, és egy több millió adatból álló vektor esetén that we no likey

**Még dolgok ebben a fejezetben amik lehet fontosak de nem nagyon érdekel**

* Beépített tömbbel való dolgok
* Különböző iterátortípusok
* Általánosan összefoglalása a standard library-s tárolóknak.

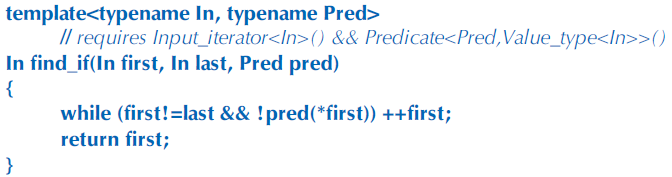
**Chapter 21. – I hate my life**

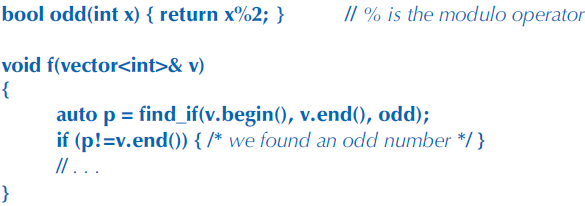
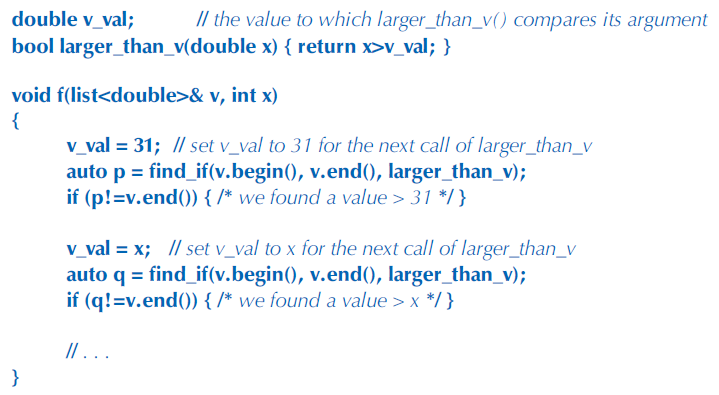
Szóval elég sok standard library algoritmus van, melyek legtöbbje az előbbiekben tanult (lmao) Iterátorokat használja.

**Find**

Legegyszerűbb algoritmus, iterátorok segítségével végigmegy az adott tárolón, és megkeres egy adott értéket.   
Igazán csodálatos

**Find\_if**

Ugyanaz mint a Find, csak nem konkrét értéket keresünk, hanem egy adott feltételt teljesítő dolgot keresünk. Viszont ezt a feltételt hogy adjuk meg?  
Így

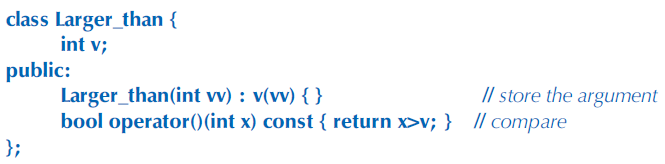
A pred, egy függvény ami igazat, vagy hamisat ad vissza, és enélkül nem működik a find\_if  
Könnyen tudunk specifikus feladatokra írni ilyen függvényeket. Ehhez, viszont minden egyes feltételre, amit szeretnénk használni, lényegében írnunk kell egy függvényt. Tehát ha írtunk egy feltételt, hogy egy szám nagyobb-e mint 19, az szép és jó, de mivan ha kisebnek kéne lennie? Vagy más a szám?

Technikailag, így meg lehet oldani, hogy írunk egy általános függvényt, aminek van egy értéke, amit majd amikor kell használni megadjuk. Nem adhatunk meg ilyen helyen egy értéket zárojelben a függvénynek, mert akkor meghívnánk a függvényt, és annak a visszatérési értéke lenne az, amit elkezdene használni a find\_if, and thats no bueno  
Összességében erre a megoldásra könyv válasza az, hogy aki ilyen kódot ír, az kb felnégyelést érdemel, szóval ezt asszem engedjük el. A megoldás, hogy kell egy objektum, amiben tartjuk az adott értéket, és egy függvényt.

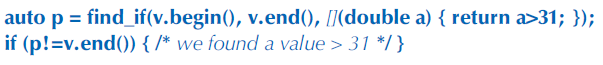
**Függvény Objektumok – bármik is legyenek**

Szóval, mit is szeretnénk? Azt hogy így nézzen ki a find\_if

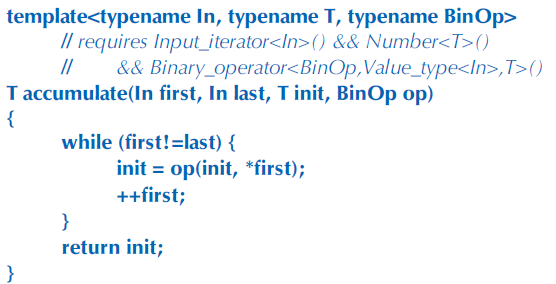
Tehát a Larger\_than, egy predikátum, amit amit meg kell tudnunk hívni a find\_if-en belül, és emellett tudnia kell tárolnia is adatot, jelen esetben a 31-es számot, hogy annál nagyobbat keresünk. Na erre a megoldás a Függvény Objektum

Itt ez a csoda, ami teljesíti a két dolgot, tudja tárolnia a predikátumot, azaz függvényt, illetve tud tárolni egy változót.  
Szóval ez egy módszer arra, hogy egy függvény magával „hordozza” az adatot, amire szüksége van.

**Lambda kifejezés ehelyett**

Az első gondolatom, amikor a find\_if-hez értünk, ez volt, hogy miért nem használunk csak egyszerűen egy lambda kifejezést, és egy pár oldallal később a könyv is eljutott idáig.

Egészen csodás.

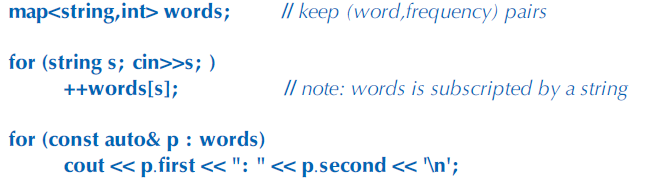
**Numerikus algoritmusok**

**Accumulate**

Mivel az iterátorok segítségével könnyedén végigtudunk mennyi egy adott tároló elemein, könnyen tudunk végezni numerikus dolgokat. Ennek az első változata a könyvben csak egész számokra jó, de ezután generalizálja ezt is, hogy ne csak egész számokra működjön. Csodás

**Inner\_product 🡪 Elképesztő mennyire nem érdekel, és nem hoz semmi újat**

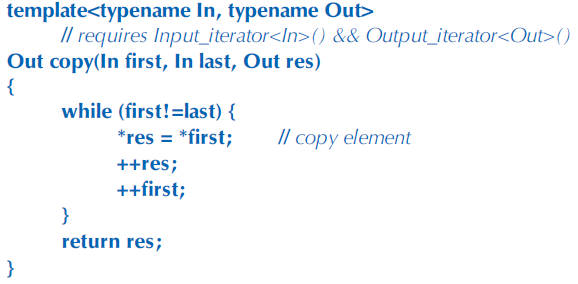
**Map mint tároló típus**

A map, lényegében adatpárok tárolására képes, típustól független. A könyv telefonkönyvet hoz fel példaként 🡪 Név, telefonszám  
De az adott példában arra használjuk az adatpárt, hogy megnézzük egy addott szót hányszor adott meg a felhasználó. Ezt jelenti a ++words[s]. Ha a szó, amit megadtunk nincs benne a map-ban, belerakja, a default int értékkel, azaz 0-val, és azt megnöveli 1-el, mivel ++.   
Ha már benne van, akkor a words-ben lévő int értéket növeli 1-el, azaz azt mutatja meg hányszor fordult elő egy szó.   
Az adatpár első tagjára words.first, míg a második tagjára words.second módon tudunk hivatkozni.

A Standard Library map egy Bináris Keresőfa, specifikusan egy piros-fekete fa.

**További típusok, amik szerintem mérsékelten kellenek vizsgára**

* Unordered Map 🡪 Hash Tábla implementáció
* Set 🡪 Egy érték nelküli Map, azaz csak 1 érték van, lényegében egy BST-ként van implemntálva

**Vissza az Algoritmusokhoz**

**Copy**

Az iterátorok segítségével a másolás egyik tárolóból a másikba szintén egy elég könnyű feladat, és ez volt drillben is.   
Van Copy-if is, ami kísértetiesen hasonlít a find\_if-re

**Sort**

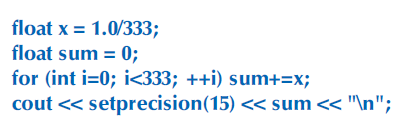
Ahogy a neve is mutatja, rendez duh  
Nem tudok ehhez többet hozzátenni

**Chapter 24. – AAAAAAAAAAAAAAAAAAAAAAAA**

A matematikai számítások a programozási szféra egy tekintélyes részét teszik ki, és mindennek az alapjai, úgyhogy erről fog szólni nagyjából ez a fejezet.

**Pontosság**

Folyamatosan integerekkel, double-ekkel foglalkozunk a programozás során, azt gondolva ezek pontos, matematikailag helyes eredményt fognak adni. Ezt nagyrészt igaz, azonban nem szabad azt elfeljeteni, hogy az integer nem az egész számokat jelentik, hanem az egész számok matematikai fogalmának egy megközelítését, míg a float és double típusok a valós számok egy megközelítése. Ezzel legtöbbször nem találkozunk, de tisztában kell lenni vele.

Például:   
Az adott programot lefuttatva azt gondolnánk, hogy a sum értéke 1 lesz igaz?  
Hát, nemigazán  
Az eredmény:

Nem sokszor találkozunk vele, de itt tökéletesen látszik, ezek a változótípusok csak megközelítései, leképezései a matematikai koncepcióknak.

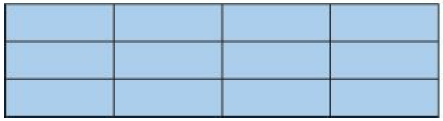
Emellett, mivel ezeket a típusokat valahogy tárolni kell, minden típusnak van egy fizikai maximális mérete. Mivel egy int-et 4 byte-on tárolunk a legnagyobb egész szám amit tudunk kezelni az ez :2147483647. Na ha mondjuk ezt még meg akarnánk szorozni valamivel, akkor már gondjaink lennének és ilyen csodálatos eredményeket kaphatnánk, mint ennél a példánál:

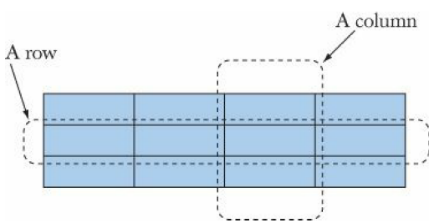
Not good.

Szóval minden változónak, az alapján, hogy mekkora memóriaterületen tároljuk, van egy elméleti legnagyobb értéke.

**Tömb**

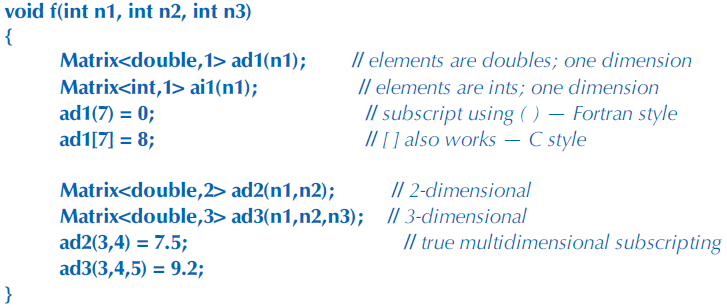
A legegyszerűbb, leggyorsabb, és legkevesebb funckióval rendelkező adatszerkezet. Csak elemek sorozata a memóriában. A vektorhoz nagyban hasonlít, de az sokkal jobban szabályozott. A Tömb, és a Vektor is úgymond 1 dimenziós. Azaz így néz ki:

Azonban vannak több dimenziós tömbök is melyek így néznek ki:   
Ezeket mátrixoknak is nevezzük.

**Mátrix**

Szóval, a Mátrix egy több dimenziós tömb, melyet sorokra,és oszlopokra bontunk.

Továbbiakban a mátrix library-vel fogunk foglalkozni.

Amikor egy mátrixot definiálunk, akkor megadjuk a típusát, valamint azt, hogy hány dimenziós. Alapvetően ez a mátrix library sorfolytonosan reprezentált a memóriában. Szóval amikor egy mátrixra hivatkozunk akkor (sor, oszlop) módon kell.

1 dimenziós mátrixoknak őszintén nem tudom mi az értelme, de valami biztos van.

Van több műveletünk, ha a egy mátrix, akkor a.size()-al visszakapjuk az elemek számát, a.dim1()-el visszakapjuk az első dimenziójában lévő elemek számát, az a.data() az az első eleme a mátrixnak a memóriában, illetve az a.slice(I,n) segítségével a[i]-től a[i+n-1]-ig kapjuk meg az elemeket.

Beszél 2D és 3D Mátrixokról, de sok újat nem mond, majd elkezd beszélni arról mire jók a mátrixok 🡪 Matek bazikálisan

* Lineáris egyenletrendszerek megoldása
* Gauss elimináció
* ****Pivoting, bármi is legyen az.

**Random számok**

Vannak random számok?  
Nem, nincsenek.

**Matematikai függvények**🡪

**Komplex Számok**

Vannak.

**Chapter 27. – End my misery**

Szóval, C++, de miért C++?  
Azért mert a C osztályokkal. Ez a fejezet a C és C++ közötti különbségekről szól.

**Kompatibilitás**

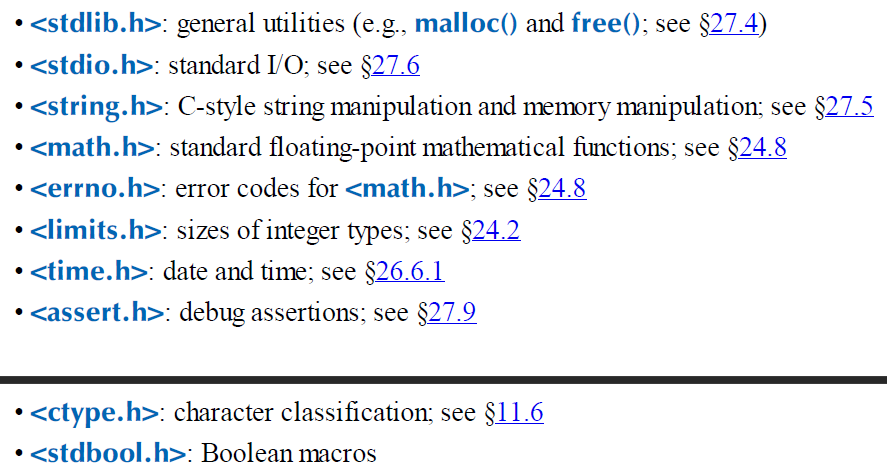
Nehéz eldönteni, hogy minek nevezzük a C++-t,de alapvetően egy elég különálló dolog, ami a C-t akarta az OOP-val összekötni. Ennek egy hozadéka, hogy van cross-compatibility, az a kód, ami működik C-ben, az C++-ban is. Visszafelé ez nem igaz.

**Mik vannak a C++-ban, amik nincsenek a C-ben?**

C++ alatt a C megfelelője.

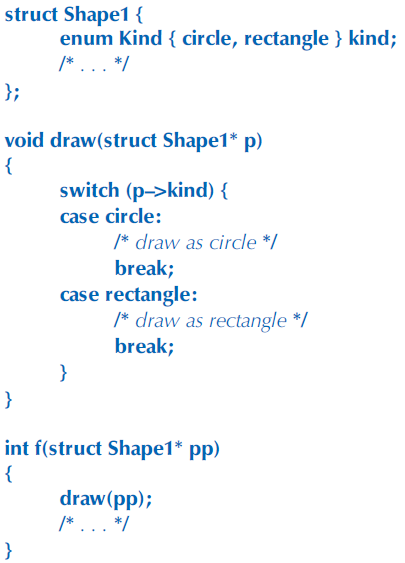
* Osztályok és tagfüggvények
  + Struct, és global függvények
* Alosztályok, virtual függvények
  + Struct, global függvény, függvényekre mutató pointerek
* Templatek
  + Macrok
* Kivételek
  + Error kódok, return value
* Függvény felülírás
  + Minden függvénynek más név
* New/delete memória
  + Malloc/free
* Referencia
  + Pointer
* Const, constexpr , const függvények
  + Macrok
* Bool
  + Intek: 0🡪 hamis - minden más egész szám 🡪 Igaz

**C Standard Library**

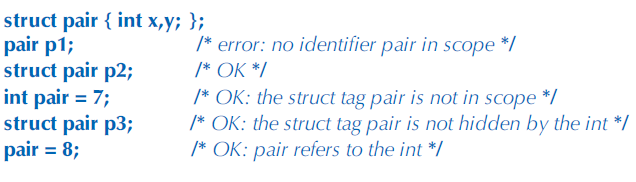
Mivel rengeteg C++ Standard library dolog osztályokon alapul, ezért C-ben nincsen:  
Vector, Map, Set, string, Stl algoritmusok: sort, find,copy, iostream,  
C libraryk:

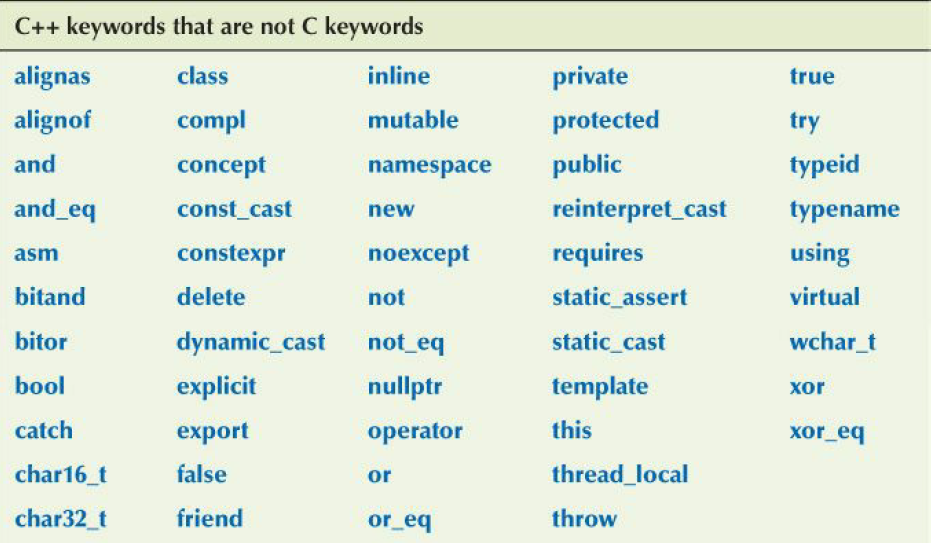
**Függvények C-ben**

* Függvénynevek egyediek
* Függvényargumentum típus nem mindig van ellenőrizve
* Nincs referencia
* Nincsenek tagfüggvények
* Más a definiálás
* Nincs függvény felülírás
* Lehet hívni úgy függvény, hogy nem is definiáltuk (lmao)

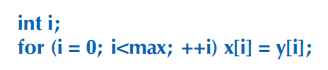
**Pointerek függvényekre 🡪**

**Nyelvi Különbségek**

Structnál, ha példányt készítünk, így kell:

**C++ Kulcsszavak amik nincsenek C-ben**

Ezek közül néhányat C-ben Macrokkal érünk el.

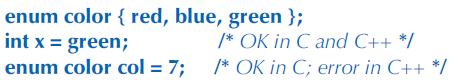
**Definiálás**

Legszembetűnőbb példa, hogy for ciklusban mondjuk nem lehet definiálni. Így kell.

C-ben helyes, C++-ban helytelen.

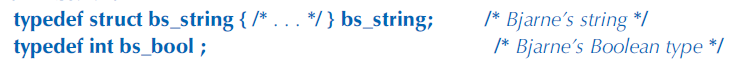
**Castolás**

Van valami különbség, de I’ve no idea

**Enum**

C-ben lehet enum-nak int értéket adni.

**Namespace**

Nincs C-ben. Mást használnak helyette.   
Elnevezik lényegében.

**Memóriakezelés**

Nem new, és delete van, hanem malloc és free, illetve még néhány más dolog

**C Stringek**

Nincsenek. Karaktertömbök vannak C-ben, nincs külön string típus. Ezért nem tudunk egyszerűen összehasonlítani sem szövegeket.

**I/O**

Nincs iostream C-ben, szóval az stdio.h-ban lévő stdin és stdout tudjuk használni.   
Legerlterjedtebb output függvény a printf(). A printf-ben viszont sok csoda van, adhatunk meg neki plusz argumentumokat, hogy milyen típusű adatot akarunk kiíratni.  
Legelterjedtebb input🡪 Scanf, Gets

**Filok**

fopen(), és fclose()

**Konstansok és Macrók**

C-ben nem használunk konstansokat, hanem előre definiált dolgokat, Makrókat.  
Ehelyett ez

**Intrusive Adattárolók**

Nincs már erőm le meg le is szarom