C++
Gyakorlat jegyzet

A jegyzetet Umann Kristóf készítette Horváth Gábor gyakorlatán.

Tartalomjegyzék

1. Bevezető

A C++ többek között a hatákonyságáról is híres. Andrei Alexandrescu azt nyilatkozta, hogy amikor a Facebooknál a backend kódján 1%ot sikerült optimalizálni, több mint 10 évnyi fizetését spórolta meg a cégnek havonta csak az áramköltségen. Nem használ garbage collectort: nincs nem várt szünet a program végrehajtásában a menedzselt nyelvekkel szemben.
A C++-szal kapcsolatban az egyik gyakori tévhit, hogy egy alacsony szintű nyelvről van szó. Bár a nyelv lehetőséget biztosít arra, hogy alacsony szinten hozzáférjünk a hardvereinkhez, számos gazdag absztrakciós lehetőséget tartalmaz. Ezeknek a használatával magas szintű kód írására is kíválóan alkalmas. A legtöbb nyelvhez képest abban emelkedik ki, hogy a C++ nyelvben ezeknek az absztrakcióknak ritkán van futási idejű költsége. Legtöbbször a fordítóprogram teljesen el tudja tüntetni ezeket az absztrakciókat a programból a fordítás során.
A C++ filozófiájának fontos eleme, hogy ha nem használunk egy adott nyelvi eszközt, akkor annak ne legyen hatása a program teljesítményére.
Fontos, hogy a C++ alapvetően nem egy objektum orientált nyelv. Bár számos nyelvi eszköz támogatja az objektum orientált stílusú programozást, de a nyelv kíválóan alkalmas más paradigmák használatára is. A funkcionális programozástól a generatív programozáson át a deklaratív stílusig sok programozási stílusra alkalmas. A nyelv nem próbál ráerőltetni egy megközelítést a programozóra, ellenben próbál minél gazdagabb eszköztárat biztosítani, hogy a megfelelő problémát a lehető legmegfelelőbb módon lehessen megoldani. Még akkor is, ha ez a különböző paradigmák keverését vonja maga után. Ezért ezt a nyelvet gyakran multiparadigmás programozási nyelvnek szokták hívni.

Cél: a tárgy során kialakítani a nyelvvel kapcsolatban egy intuíciót, ami segítségével elkerülhetőek alapvető hibák is. Az előzménytárgyakban az egyszerűség kedvéért gyakran féligazságok hangzottak el, ezeket is helyre kell rakni.

1.1. Mi az a C++?

Alapvetően a nyelv két összetevőből áll. Az aktuális szabványból és annak implementációiból (fordítók + szabványkönyvtárak). A szabvány, ami meghatározza a nyelv nyelvtanját, valamint a szemantikát: mit jelentenek a leforduló programok (nem definiál minden részletet). Emellett a szabvány definiálja a szabványkönyvtárat is, amit minden szabványos C++ fordító mellé szállítani kell. Az első C++ szabvány a C++98 volt. További szabványai: C++03, C++11, C++14, C++17.

A szabvány alapján számos fordító (implementáció) létezik a C++ kódok fordítására: MSVC (Visual Studio), GCC, Clang. Létezik számos fejlesztői környezet is, mint például: CLion, QtCreator, CodeBlocks, VIM. De ezek nem fordítók, legtöbbször a fent említett fordítók közül használnak egyet.

2. Különböző viselkedések kategorizálása

Egy reménytelen megközelítés lenne a szabványban minden szintaktikusan (nyelvtanilag) helyes kódhoz pontos szemantikát (működést) társítani. Ennek mind elméleti és gyakorlati oka van. Ezért a C++ szabvány néhány esetben nem vagy csak részben definiálja egy adott program működését. A következőkben erre fogunk példákat látni.

2.1. Nem definiált viselkedések

```
int main()
{
    int i = 0;
    std::cout << i++ << i++ << std::endl;
}
```

Lehetséges kimenet: `01` (GCC 6.1 fordítóval 64 bites x86 Linux platformon)
Lehetséges kimenet: `10` (Clang 3.9 fordítóval 64 bites x86 Linux platformon)

Fordítás és futtatás után különböző eredményeket kaphatunk, mert itt az, hogy mikor értékelődik ki a két `++i` a kifejezésen belül, az **nem specifikált.** Ha a szabvány nem terjed ki arra, hogy milyen viselkedésű kódot generáljon a fordító, akkor a fordító bármit választhat.

Gyakran eldönthetetlen előre, hogy mikor mi lesz a leghatékonyabb megoldás, ez az egyik ok, hogy nem definiál mindent a szabvány.

Ez lehetőséget ad a fordítónak arra, hogy **optimalizáljon.**

A C++ban van un. szekvenciapontok, és a szabvány csak azt mondja ki, hogy a szekvenciapont előtti kód hamarabb kerüljön végrehajtásra mint az utána levő. Mivel itt az i értékadása után és csak az `std::endl` után van szekvenciapont, így az, hogy milyen sorrendben történjen a kettő közötti kifejezés részkifejezéseinek a kiértékelése, az a fordítóra van bízva.

A C++ban nem meghatározott, hogy két szekvenciapont között mi az utasítások végrehajtásának a sorrendje.

Az, hogy két részkifejezés szekvenciaponttal történő elválasztás nélkül ugyanazt a memóriaterületet módosítja, **nem definiált** viselkedést eredményez. Nem definiált viselkedés esetén a fordító vagy a futó program bármit csinálhat. A szabvány semmiféle megkötést nem tesz.

2.1.1. Megjegyzés. Az a program, amely nem definiált viselkedéseket tartalmaz, hibás.

2.2. Nem specifikált viselkedések

Amennyiben a szabvány definiál néhány lehetséges opciót, de a fordítóra bízza, hogy az melyiket választja, akkor **nem specifikált** viselkedésről beszélünk.
A nem specifikált viselkedés csak akkor probléma, ha a program végeredményét (megfigyelhető működését) befolyásolhatja a fordító választása. Például a fenti kódot módosíthatjuk a következő képpen:

```
int main()
{
    int i = 0;
    int j = 0;
    std::cout << ++i << ++j << std::endl; // 11
}
```

Bár azt továbbra se tudjuk, hogy `++i` vagy `++j` értékelődik ki hamarabb, (*nem specifikált*), azt biztosan tudjuk, hogy 11-et fog kiírni (a program végeredménye *jól definiált*).

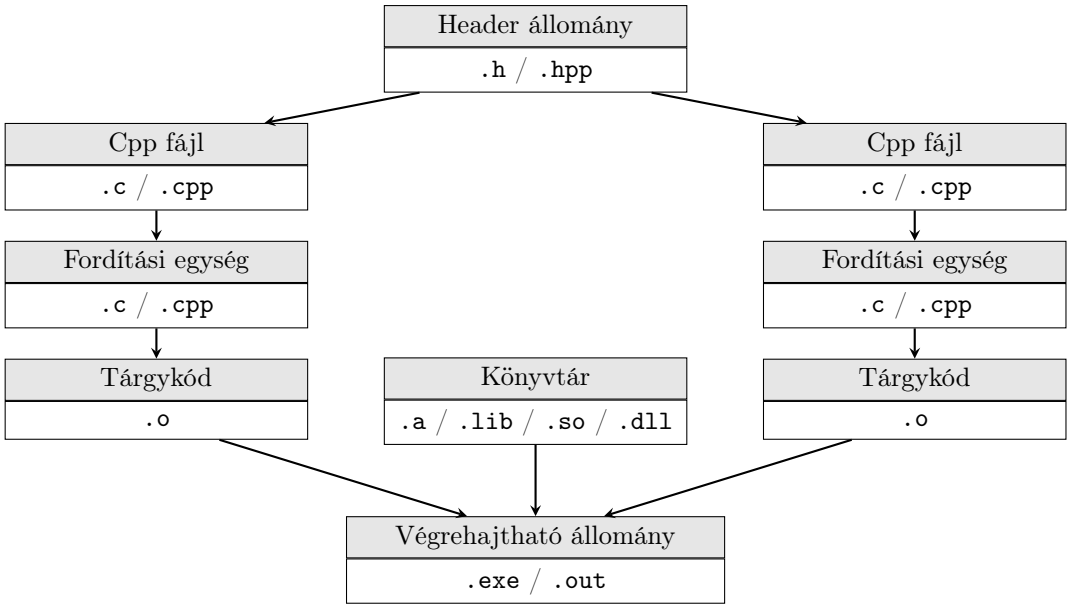
2.3. Implementáció által definiált viselkedés

A szabvány nem köti meg, hogy egy `int` egy adott platformon mennyi byte-ból álljon. Ez állandó, egy adott platformon egy adott fordító mindig ugyanakkorát hoz létre, de platform/fordítóváltás esetén ez változhat. Ennek az az oka, hogy különböző platformokon különböző választás eredményez hatékony programokat. Ennek köszönhetően hatékony kódot tud generálni a fordító, viszont a fejlesztő dolga, hogy megbizonyosodjon róla, hogy az adott platformon a primitív típusok méretei megfelelnek a program által elvárt követelményeknek.

3. A fordító működése

A fordítás 3 fő lépésből áll:

1. Preprocesszálás
2. Fordítás (A tárgykód létrehozása)
3. Linkelés (Szerkesztés)



Szürkében az adott fordítási lépés neve, alatta az így létrehozott fájl kiterjesztése (leggyakrabban).

A fordítás a preprocesszor parancsok végrehajtásával kezdődik (például a **header fájl**ok beillesztése a **cpp fájl**okba), az így kapott fájlot hívjuk **fordítási egység**nek (*translation unit*). A fordítási egységek különkülön fordulnak **tárgykód**dá (*object file*). Ahhoz hogy a tárgykódokból **futtatható állomány**t (*executable file*) lehessen készíteni, össze kell linkelni őket. A saját forráskódunkból létrejövő tárgykódok mellett a linker a felhasznált könyvtárak tárgykódjait is bele fogja szerkeszteni a végleges futtatható állományba.

A következő pár szekcióban megismerjük a fenti 3 lépést alaposabban.

3.1. Preprocesszálás

A preprocesszor (vagy előfeldolgozó) használata a legtöbb esetben kerülendő. Ez alól kivétel a header állományok include-olása. A preprocesszor **primitív** szabályok alapján dolgozik és **nyelvfüggetlen.** Mivel semmit nem tud a C++-ról, ezért sokszor a fejlesztő számára meglepő viselkedést okozhat a használata. Emellett nem egyszerű diagnosztizálni a preprocesszor használatából származó hibákat. További probléma, hogy az automatikus refaktoráló eszközök használatát is megnehezíti a preprocesszor túlhasználata.
A következőkben néhány preprocesszor direktívával fogunk megismerkedni. Minden direktíva `#` jellel kezdődikcl. Ezeket a sorokat a fordító a program fordítása szempontjából figyelmen kívül hagyja.

alma.h

```
#define ALMA 5

ALMA ALMA ALMA
```

A `#define ALMA 5` parancs azt jelenti, hogy minden `ALMA` szót ki kell cserélni a fájlban `5`-re.
Az előfeldolgozott szöveget a `cpp alma.h` parancs kiadása segítségével tekinthetjük meg.
Az így kapott fájlból kiolvasható előfeldolgozás eredménye: `5 5 5`.

alma.h

```
#define KORTE

#ifdef KORTE
    MEGVAN
#else
    KORTE
#endif
```

A fent leírtakon kívül a `#define` hatására a preprocesszor az első argumentumot makrónak fogja tekinteni. A fenti kódban rákérdezünk, hogy ez a `KORTE` makró definiálva van-e (az `#ifdef` paranccsal), és mivel ezt fent megtettük, `#else`-ig (vagy annak hiányában `#endif`-ig) minden beillesztésre kerül, kimenetben csak annyi fog szerepelni, hogy `MEGVAN`.

alma.h

```
#define KORTE
#undef KORTE

#ifdef KORTE
    MEGVAN
#else
    KORTE
#endif
```

Az `#undef` paranccsal a paraméterként megadott makrót a preprocesszor nem tekinti továbbá makrónak, így a kimenetben `KORTE` lesz.
Látható, hogy az előfeldolgozót kódrészletek kivágására is lehet használni. Felmerülhet a kérdés, ha az eredeti forrásszövegből az előfeldolgozó kivág illetve beilleszt részeket, akkor a fordító honnan tudja, hogy a hiba jelentésekor melyik sorra jelezze a hibát? Hiszen az előfeldolgozás előtti és utáni sorszámok egymához képest eltérnek. Ennek a problémának a megoldására az előfeldolgozó beszúr a fordító számára plusz sorokat, amik hordozzák azt az információt, hogy a feldolgozás előtt az adott sor melyik fájl hányadik sorában volt megtalálható.

3.1.1. Megjegyzés. A fordítás közbeni ideiglenes fájlokat a `g++ -save-temps hello.cpp` paranccsal lehet lementeni.

A már bizonyára ismerős `#include` egy paraméterént megadott fáj tartalmát illeszti be egy az egyben az adott fájlba, és így nagyon jelentősen meg tudják növelni a kód méretét, ami a fordítást lassítja. Ezért óvatosan kell vele bánni.

pp.h

```
#include "pp.h"
```

Rekurzív include-nál, mint a fenti példában, az előfeldolgozó egy bizonyos mélységi limit után leállítja a preprocesszálást.
Sok és hosszú include láncok esetén azonban nehéz megakadályozni, hogy kör kerüljön az include gráfba, így akaratlanul is a rekurzív include-ok aldozatai lehetünk.

pp.h

```
#ifndef _PP_H_
#define _PP_H_

```

```
    FECSKE

#endif
```

alma.h

```
#include "pp.h"
#include "pp.h"
#include "pp.h"
#include "pp.h"
#include "pp.h"
```

Egy trükk segítségével megakadályozhatjuk azt, hogy többször be legyen illesztve `FECSKE`. Először megnézzük, hogy `_PP_H_` szimbólum definiálva van-e. Ha nincs, definiáljuk. Mikor legközelebb ezt meg akarnánk tenni (a második `#include "pp."` sornál), nem illesztjük be a `FECSKE`-t, mert `#ifndef _PP_H_` kivágja azt a szövegrészt. Ez az úgy nevezett **header guard** vagy **include guard.**

A preprocesszor az itt bemutatottaknál sokkal többet tud, de általában nem érdemes túlhasználni a fent említett okok miatt.

3.2. Linkelés

fecske.cpp

```
void fecske() {}
```

main.cpp

```
int main()
{
    fecske();
}
```

Ez nem fog lefordulni, mert vagy csak a main.cpp-ből létrejövő fordítási egységet, vagy a fecske.cpp-ből létrejövő fordítási egységet látja a fordító, egyszerre a kettőt nem. Megoldás az ha **forward deklarálunk**, `void fecske();`-t beillesztjük a main függvény fölé, mely jelzi a fordítónak, hogy a `fecske` az egy függvény, `void` a visszatérési értéke és nincs paramétere.

Ekkor `g++ main.cpp` paranccsal történő fordítás a linkelési fázisánál kapunk hibát, mert nem találja a `fecske` függvény definícióját. Ezt ahogy korábban láttuk, úgy tudjuk megoldani, ha `main.cpp`-ből és `fecske.cpp`-ből is tárgykódot készítünk, majd összelinkeljük őket. `main.cpp`-ben lesz egy hivatkozás egy olyan `fecske` függvényre, melynek `void` a visszatérési értéke és paramétere nincs, és `fecske.cpp` fogja tartalmazni e függvény definícióját.

```
g++ -c main.cpp
g++ -c fecske.cpp
```

A fenti paranccsal lehet tárgykódot előállítani.

```
g++ main.o fecske.o
```

Ezzel a paranccsal pedig az eredményül kapott tárgykódokat lehet összelinkelni. Rövidebb, ha egyből a cpp fájlokat adjuk meg a fordítónak, így ezt a folyamatot egy sorral letudhatjuk..

```
g++ main.cpp fecske.cpp
```

Ha a `fecske.cpp`-ben sok függvény van, akkor nem célszerű egyesével forward deklarálni őket minden egyes fájlban, ahol használni szeretnénk ezeket a függvényeket. Ennél egyszerűbb egy header fájl megírása, amiben deklaráljuk a `fecske.cpp` függvényeit.

fecske.h

```
#ifndef _FECSKE_H_
#define _FECSKE_H_
    void fecske();
#endif
```

Ilyenkor elég a `fecske.h`-t includeolni.

Függvény definíciónak nevezzük azt, amikor megmondjuk a függvénynek hogy mit csináljon. Ez egyben deklaráció is, hiszen a paraméterekről és visszatérési értékekről is tartalmazza a szükséges információkat.

Függvény deklarációnak nevezzük azt, amikor függvény használatáról adunk információt. A paraméterek típúsáról, visszatérési értékről és a függvény nevéről.

Szokás a fecske.h-t a fecske.cpp-be is includeolni, mert ha véletlenül ellent mondana egymásnak a definíció a cpp fájlban és a deklaráció a header fájlban akkor a fordító hibát fog jelezni. (Például ha eltérő visszatérési érték típust adtunk meg a definíciónak a C++ fájlban és a deklarációnak a header fájlban.)
Valami akárhányszor deklarálhatunk, azonban ha a deklarációk ellentmondanak egymásnak, akkor fordítási hibát kapunk. Definiálni viszont mindent pontosan egyszer kell. Több definíció vagy a definíció hiánya problémát okozhat. Ezt az elvet szokás **One Definition Rule**-nak, vagy röviden **(ODR)**-nek hívni.

fecske.h

```
#ifndef _FECSKE_H_
#define _FECSKE_H_
    void fecske();
    int macska() {}
#endif
```

Ha több fordítási egységből álló programot fordítunk, melyek tartalmazzák a `fecske.h` headert, akkor a preprocesszor több macska függvény definíciót csinál, és linkeléskor a linker azt látja, hogy egy függvény többször van definiálva, és ez linkelési hibát eredményez.

3.2.1. Megjegyzés. A header fájlokba nem szabad definíciókat rakni (bár kivétel létezik, pl. template-ek, inline függvények, melyekről később lesz szó).

4. Figyelmeztetések

A fordító gyanús vagy hibás kódrészlet esetén tud figyelmeztetéseket generálni. A legtöbb fordító alapértelmezetten elég kevés hibalehetőségre figyelmeztet. További figyelmeztetések bekapcsolásával hamarabb, már fordítási időben megtalálhatunk bizonyos hibákat vagy nem definiált viselkedéseket. Ezért ajánlott a `-Wall`, `-Wextra` kapcsolókat használni.

```
g++ -Wall -Wextra hello.cpp
```

5. Optimalizálás

A fordításnál bekapcsolhatunk optimalizációkat, a GCC-nél pl. így:

```
g++ hello.cpp -O2
```

Az `-O2` paraméter a kettes szintű optimalizációk kapcsolja be. Alapértelmezetten nincs optimalizáció (`-O0`), és egészen `-O3`-ig lehet fokozni azt.

hello.cpp

```
int factorial(int n)
{
    if (n <= 0) return 1;
    else return n*factorial(n-1);
}

int main()
{
    std::cout << factorial(5) << std::endl;
}
```

A `g++ -save-temps hello.cpp` paranccsal fordítva a temporális fájlokat is meg tudjuk nézni – hello.s lesz az assembly fájl neve, mely a fordító a kódunk alapján generált. Kiolvasható benne ez a két sor:

```
movl    $5, (%esp)
call    __Z9factoriali
```

5.0.1. Megjegyzés. Az, hogy a fordító milyen assembly kódot alkot az input fájlból, implementációfüggő, ebben az esetben ezt az eredményt kaptuk.

Látható, hogy a `factorial` függvény 5 paraméterrel meg lett hívva (az hogy pontosan itt mi történik, az lényegtelen).

Amennyiben azonban `g++ -save-temps hello.cpp -O2` paranccsal fordítunk, az optimalizált assembly kódból kiolvasható, hogy a kód (kellően friss gcc-vel) a faktoriális kiszámolása helyett a végeredményt (120at) tartalmazza.

```
movl    $120, (%esp)
```

Így, mivel az eredmény már fordítási időben kiszámolásra került, futási időben nem kell ezzel plusz időt tölteni. A fordító sok ehhez hasonló **optimalizációt** végez. Ennek hatására a szabványos és csak definiált viselkedést tartalmazó kód jelentése nem változhat, viszont sokkal hatékonyabbá válhat.

5.0.2. Megjegyzés. `-O3` Olyan optimalizálásokat is tartalmazhat, amik agresszívabban kihasználják, ha egy kód nem definiált viselkedéseket tartalmaz, míg az`-O2` kevésbé aggresszív, sokszor a nem szabványos kódot se rontja el. Mivel nem definiált viselkedésekre rosszul tud reagálni az `-O3`, így néha kockázatos használni.

6. Globális változók

6.1. Féligazságok előzménytárgyakból

Előzménytárgyakból azt tanultuk, hogy a program futása a main függvény végrehajtásával kezdődik. Biztosan igaz ez?

```
std::ostream& os = std::cout << "Hello";
int main()
{
    std::cout << "valami";
}
```

Kimenet: `Hellovalami`.

Tehát ez nem volt igaz. A program végrehajtásánál az első lépés az un. **globális változók** inicializálása.
Ennek az oka az, hogy a globális változók olyan objektumok, melyekre a program bármely pontján hivatkozni lehet, így ha `os`-t akarnám használni a `main` függvény első sorában, akkor ezt meg lehessen tenni. Inicializálatlan változó használata pedig nem definiált viselkedés, ezért fontos már a `main` végrehajtása előtt inicializálni a globálisokat.

```
int f()
{
    return 5;
}

int x = f();

int main()
{
    std::cout << "valami";
}
```

Itt szintén az `f()` kiértékelése a `main` függvény meghívása előtt történik, hogy a globális változót létre lehessen hozni.

6.2. Globális változók definíciója és deklarációja

Globális változókat úgy tudunk létrehozni, hogy közvetlen egy névteren belül (erről később) definiáljuk őket.

main.cpp

```
int x;

int main() {}
```

`x` egy globális változó. Azonban mit tudunk tenni, ha nem csak a `main.cpp`-ben, hanem egy másik fordítási egységben is szeretnénk rá hivatkozni?

other.cpp

```
int x;

void f()
{
    x = 0;
}
```

Sajnos ha `main.cpp`-t és `other.cpp`-t együtt fordítjuk, fordítási hibát kapunk, ugyanis megsértettük az ODR-t, hiszen `x` kétszer van definiálva. Ezt úgy tudjuk megoldani, ha `x`-et forward deklaráljuk az `extern` kulcsszóval!

other.cpp

```
extern int x;

void f()
{
    x = 0;
}
```

Csupán annyi a fontos, hogy `x`-et valamikor definiálni is kell (mely jelenleg a `main.cpp`-bentalálható).

6.2.1. Megjegyzés. A globális változók deklarációit érdemes külön header fájlba kigyűjteni.

6.3. Globális változók inicializációja

A globális változók egyedi módon kapnak kezdőértéket (inicializálódnak). Amennyiben egy nem globális `int`-et hozunk létre és nem adunk neki kezdőértéket, annak értéke nem definiált lesz (memóriaszemét).

```
int i;

int main()
{
    std::cout << i << std::endl; // 0
}
```

Azonban mégis mindig 0-t fog ez a program kiírni. Ennek oka az, hogy a globális változók mindig 0-ra inicializálódnak (legalábbis az `int`-ek). A globális változókat csak egyszer hozzuk létre a program futásakot, így érdemes jól definiált kezdőértéket adni neki.
Azonban a stacken (mellyel hamarosan megismerkedünk) rengetegszer létre kell hozni változókat, nem csak egyszer, így ott nem éri meg minden alkalommal egy jól definiált kezdőértékkel inicializálni. Sokkal nagyobb lenne a hatása a futási időre.
Annak, hogy miért épp 0-ra inicializálódnak a globális változók, az az oka, hogy ezt a modern processzorok gyorsan tudják kivitelezni minden platformon.

6.4. Problémák a globális változókkal

A linkelés vajon befolyásolhatja a program megfigyelhető viselkedését?

main.cpp

```
std::ostream& o = std::cout << "Hello";
int main() {}
```

fecske.cpp

```
std::ostream& o2 = std::cout << " World";
```

Itt nem specifikált a két globális változók inicializációs sorrendje, és ha más sorredben linkeljük a fordítási egységekből keletkező tárgykódot, mást ír ki.

`g++ main.cpp fecske.cpp` $\neq$ `g++ fecske.cpp main.cpp`

6.4.1. Megjegyzés. Ez utolsó példa nem számít jó kódnak, mert nem specifikált viselkedést használ ki. A program kimenete nem definiált. Ez is egy jó elrettentő példa, miért nem érdemes globális változókat használni.

Ezen kívül számos egyéb problémát is felvetnek a globális változók: túlzott használatuk a sok paraméterrel rendelkező függvények elkerülése végett fordulhat elő, azonban gyakran így sokkal átláthatatlanabb kódot kapunk. Mivel bárhol hozzá lehet férni egy globális változóhoz, nagyon nehéz tudni, mikor hol módosul.

6.4.2. Megjegyzés. Párhuzamos programozásnál a globális változók túl az átláthatatlanságon még sokkal több fejtörést okoznak: mi van akkor, ha két párhuzamosan futó függvény ugyanazt a változót akarja módosítani? Ennek megelőzése globális változóknál ezt rendkívül körülményes lehet. A naív megoldás (kölcsönös kizárás) pedig rosszul skálázódó programot eredményez.

7. Láthatóság, élettartam

Egy objektum **láthatóságának** nevezzük a kódnak azon szakaszait, melyeknél lehet rá hivatkozni.

Egy objektum **élettartamának** nevezzük a kód azon szakaszát, melynél bent szerepel a memóriában. Amikor egy objektum élettartama elkezdődik, azt mondjuk, az objektum létrejön, míg az élettartam végén az objektum megsemmisül.

7.0.1. Megjegyzés. Ez alapján megállapíthatjuk, hogy egy globális változó láthatósága és élettartama a program futásának elejétől végéig tart.

Figyeljük meg, mikor tudunk `x` változóra hivatkozni (azaz hol lesz `x` látható)!

```
int x;

int main()
{
    int x = 1;
    {
        int x = 2;
        std::cout << x << std::endl; // 2
    }
}
```

Megfigyelhető, hogy a `main` függvény elején létrehozott `x` az utána következő blokkban teljesen elérhetetlen – nincs olyan szabványos nyelvi eszköz, amivel tudnánk rá hivatkozni. Ezt a folyamatot **leárnyékolás**nak (*shadowing*) nevezzük. Azonban a külső, globális `x`-re bármikor tudunk hivatkozni az alábbi módon:

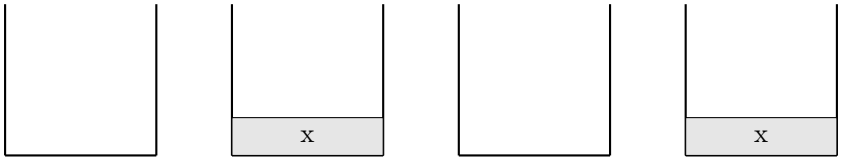
```
int x;

int main()
{
    int x = 1;
    {
        int x = 2;
        std::cout << ::x << std::endl; // 0
    }
}
```

8. A stack működése

A stack a C++ alapértelmezett tárolási osztálya lokális változók esetén: minden változó alapértelmezetten itt jön létre és semmisül meg. Az itt létrejött változók automatikusan megsemmisülnek. Az élettartamuk a definíciójuktól az adott blokk végéig tart.

```
#include <iostream>

int f()
{
    int x = 0; //x létrejön
    ++x;
    return x;
} //x megsemmisül

int main()
{
    for (int i = 0; i<5; i++)
        std::cout << f() << ' '; // 1 1 1 1 1
}
```

A fenti kód futása során a stack-et így képzelhetjük el:



Az ábrán egy stack-et látunk. Amikor a vezérlés az `f` függvényhez ér, és ott létrehozza az `x` változót, azt behelyezi a stack-be. A `return` kulcsszó hatására készít `x`-ről egy temporális példányt, ami a függvény visszatérési értéke lesz. Amikor a vezérlés visszatér a `main` függvényhez, x-re nem tudunk tovább hivatkozni, így azt megsemmisíti, és ez ismétlődik, ameddig a ciklus véget nem ér.

A stack egy FILO (*first in last out*) adatszerkezet – azaz azt az elemet „dobja” ki a vezérlés a stack-ből, melyet utoljára rakott be.

9. Paraméter átvétel

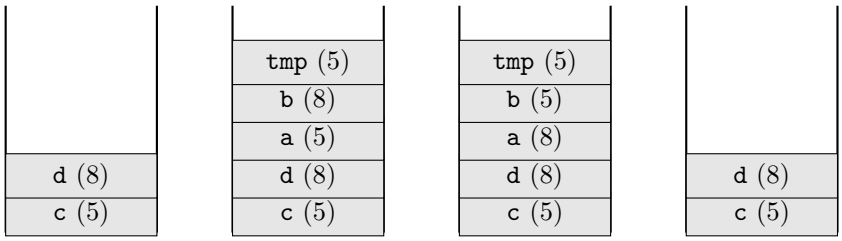
9.1. Érték szerinti paraméter átvétel

Próbáljuk megvalósítani a swap függvényt!

```
#include <iostream>
void swapWrong(int a, int b)
{
    int tmp = a;
    a = b;
    b = tmp;
}

int main()
{
    int c = 5, d = 8;
    swapWrong(c, d);
    std::cout << c << ' ' << d << std::endl;
}
```

A program kimenete `5 8`. Ez egy teljesen jól definiált viselkedés. Ennek az az oka, hogy itt **érték** szerint vettük át (*pass by value*) `a` és `b` paramétert. A következő ábrán megfigyelhetjük mi is történik pontosan. Képzeljük el, hogy a stackbe a program elrakja a `c` és `d` változókat. Eztán meghívja a `swapWrong` függvényt, melyben

létrehozott `a` és `b` paraméterek szintén a stackre kerülnek. Bár a függvényre lokális `a` és `b` paraméterek értékét megcseréli, de a függvényhívás után ezeket ki is törli a stackből. Az eredeti `c` és `d` változók éréke nem változott a függvényhívás során. C++ban alapértelmezett módon a paraméterátadás érték szerint történik.



9.2. Mutatók érték szerinti átadása

A mutatók olyan nyelvi elemek, melyek egy adott típusú memóriaterületre mutatnak. Segítségükkel anélkül is tudunk hivatkozni egy adott objektumra (és nem csak a másolatára), hogy közvetlenül az objektummal dolgoznánk. Most röviden megismerkedünk velük, de később részletesebben visszatérünk rájuk.

```
int main()
{
    int c = 5, d = 8;
    int *p = &c;
}
```

A fenti példában `p` egy mutató (*pointer*), mely egy `int` típusra mutat. Ahhoz, hogy értéket tudjunk adni egy mutatónak, egy memóriacímet kell neki értékül adni, erre való a **címképző operátor** (&). Ha a mutató által *mutatott értéket* szeretnénk módosítani, akkor dereferálnunk kell a **dereferáló operátor**ral (*).

```
int *p = &c; //referáljuk c-t
*p = 4; //dereferáljuk p-t
p = &d;
*p = 7;
```

Rendre: pointer inicializálása, pointer által mutatott érték módosítása, pointer átállítása másik memóriacímre, és a mutatott érték módosítása.
Egy mutató mutathat változóra, másik mutatóra vagy sehova. Azokat a mutatókat, melyek sehová sem mutatnak, null pointernek nevezzük, és így hozhatjuk létre őket:

```
p = 0;    p = NULL;    p = nullptr;
```

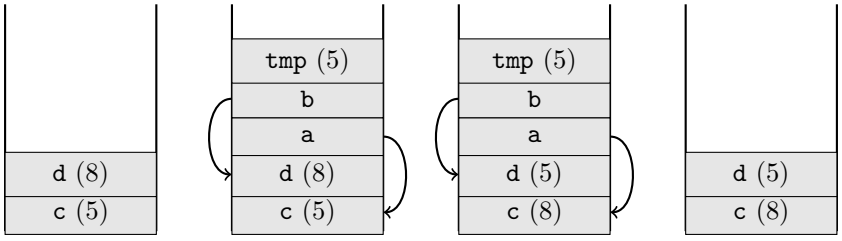
9.2.1. Megjegyzés. Ez a három értékadás (közel) ekvivalens, azonban a `nullptr` kulcsszó csak C++11ben és azutáni szabványokban érhető el.

Nézzük meg, hogy hogyan tudunk megcserélni két értéket ezúttal helyesen, mutatók segítségével.

```
void swapP(int *a, int *b)
{
    int tmp = *a;
    *a = *b;
    *b = tmp;
}
```

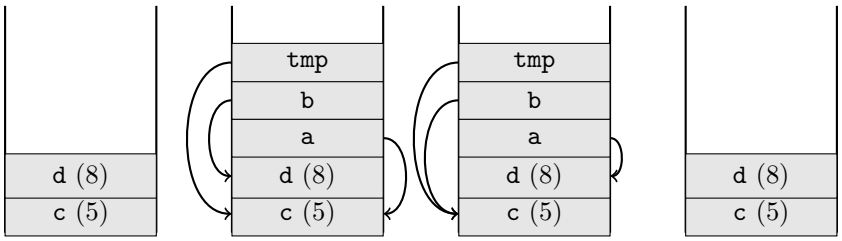
Amennyiben ezt a függvényt hívjuk meg, valóban megcserélődik a két változó értéke. De ehhez fontos, hogy ne `swapP(c, d)`-t írjunk, az ugyanis az fordítási hibához vezetne, hiszen a `c` és `d` típusa `int`, és nem `int*`. Ahhoz, hogy értéket adjunk egy pointernek, a `c`-hez és `d`-hez tartozó memóriacímeket kell átadni, így a `swapP(&c, &d)` hívás lesz megfelelő.

9.2.2. Megjegyzés. A mutatókat továbbra is érték szerint adjuk át. Az `a` és `b` paraméterekben lévő memóriacím tehát a másolata annak, amit a hívás helyén megadtunk.




```
void swapWrong2(int *a, int *b)
{
    int *tmp = a;
    a = b;
    b = tmp;
}
```

Ebben a példában nem a pointerek által mutatott értéket, hanem magukat a pointereket cseréljük meg. Itt az fog történni, hogy a függvény belsejében `a` és `b` pointer másra fog mutatni. A mutatott értékek viszont nem változnak.



9.3. Referencia szerinti paraméter átadás

Megállapíthatjuk, hogy az előző megoldásnál nem változtattuk meg, hogy mire mutassanak a pointerek, így azokat konstansként is definiálhatnánk. A konstans pointerek módosíthatják a mutatott értéket, de nem lehet őket átállítani egy másik memória címre. Úgy tudunk egy ilyen pointert létrehozni, hogy a csillag után írjuk a `const` kulcsszót.

```
void swap(int * const a, int * const b)
{
    //...
}
```

Egy kis szintaktikai cukorkával megúszhatjuk azt, hogy folyton kiírjuk a `* const`-ot (lévén nem akarjuk megváltoztatni, hogy ilyen esetben a pointer hova mutasson). Erre való a referencia szerinti paraméter átváétel (*pass by reference*). A referencia hasonlóan működik, mintha egy konstans pointer lenne, csak nem lehet sehova se mutató referenciát létrehozni.

```
void swapRef(int &a, int &b)
{
    int tmp = a;
    a = b;
    b = tmp;
}
```

Ez a függvény lényegében ekvivalens a `swapP` fügvénnyel.

9.3.1. Megjegyzés. Ez bár ezt referencia szerinti átvételnek nevezzük, de itt is történik másolás, a memóriacímet ítt is érték szerint vesszük át.

Megjegyzendő, hogy a fenti `swapRef` függvény meghívásakor nem kell jeleznünk, hogy memóriacímeket akarunk átadni, `swapRef(a,b)`-t kell írnunk.

9.3.2. Megjegyzés. Egy referenciát mindig inicializálni kell. Csak úgy mint egy konstanst (különben fordítási hibát kapunk.)

9.4. Visszatérési érték problémája

Nem primitív (pl. `int`) típusoknál gyakran megeshet, hogy egy adott típushoz tartozó pointer mérete kisebb, mint magának az objektumé, így megérheti mindentől függetlenül a paramétert referencia szerint átvenni. Ezen felbátorodva mondhatnánk azt is, hogy referenciával is térjünk vissza (a következő példában tekintsünk el attól, hogy `int`-el dolgozunk, bátran képzeljük azt hogy az pl. egy nagyon nagy mátrix)!

```
int& addOne(int &i)
{
    i++;
    return i;
}

int main()
{
    int i = 0;
    int a = addOne(i);
    std::cout << a << std::endl;
}
```

A fenti kóddal semmi gond nincs is. De mi van, ha egy picit módosítunk rajta?

```
int& addOne(int &i)
{
    int ret = ++i;
    return ret;
}
```

A baj máris megvan, amit egy warning is jelezni fog nekünk: olyan objektumra hivatkozó referenciát adunk vissza, amely `addOne`-on belül lokális. Ez azt jelenti, hogy amint a vezérlés visszatér a `main` függvényhez, `ret` megsemmisül, és a `main` függvény pedig a `ret`-hez tartozó címen lévő értéket próbálná meg lemásolni. Mivel viszont a `ret` már ezen a ponton megsemmisült, semmi nem garantálja, hogy azon a memóriaterületen ne következett volna be módosítás.

Az olyan memóriaterületre való hivatkozás, mely nincs a program számára lefoglalva, nem definiált viselkedést eredményez.

9.4.1. Megjegyzés. Értelemszerűen pointerekkel ez ugyanúgy probléma.

10. Statikus változók/függvények

A `static` kulcsszónak számos jelentése van, annak függvényében, hogy milyen kontextusban írjuk egy változó vagy függvény elé.

10.1. Fordítási egységre lokális változók

A függvényeken és osztályokon kívül deklarált statikus változók az adott fordítási egységre lokálisak – élettartamuk a futás elejétől végigtartamig tart, és kizárólagosan az adott fordítási egységben láthatóak.

main.cpp

```
#include <iostream>

static int x;

int main()
{
    x = 2;
}
```

other.cpp

```cpp
#include <iostream>

static int x;

void f()
{
    x = 0;
}
```

Ha ezt a két fájlt együtt fordítjuk, nem kapunk linkelési hibát, ugyanis a `main.cpp`-ben lévő `x` egy teljesen más változó, mint ami az `other.cpp`-ben van.

Csak úgy mint a globális változókra, fordítási egységen belül bármikor hivatkozhatunk egy statikusra, és hasonló módon inicializálódnak.

```cpp
static int x;

int main()
{
    int x = 4;
    std::cout << ::x << std::endl; // 0
}
```

10.2. Függvényen belüli statikus változók

Azokat a változókat, melyek függvényen belül vannak a `static` kulcsszóval definiálva, függvényszintű változónak is szokás hívni. Élettartamuk a függvény első hívásától a program futásának végéig tart, míg láthatóságuk csak az adott függvényen belül van. A hagyományos lokális változókkal ellenben tehát nem semmisülnek meg, amikor az adott függvény futása befejeződik. A következő kódrészlet szemlélteti ezt a viselkedést.

```cpp
int f()
{
    static int x = 0;
    ++x;
    return x;
}

int main()
{
    for(int i = 0; i<5; i++)
        std::cout << f() << ' '; // 1 2 3 4 5
}
```

Ahogy az megfigyelhető fent, `x` csak egyszer inicializálódik, majd a későbbi függvényhívások után egyre növekszik az értéke.

10.3. Fordítási egységre lokális függvények

Nem csak változókat, függvényeket is deklarálhatunk statikusnak, melyek a fordítási egységre lokálisak.

```cpp
static int f()
{
    return 0;
}

int main() {std::cout << f();} // 0
```

Ezek a függvények csak az adott fordítási egységen belül érhetőek el.

10.3.1. Megjegyzés. Figyelem! A később szóba kerülő metódusok esetében mást jelent a `static` kulcsszó, mint amit itt leírtunk.

10.4. Névtelen/anonim névterek

Fordítási egységre lokális változókat és függvényeket tudunk deklarálni névtelen névterek (*unnamed namespaces*, vagy *anonymous namespaces*) segítségével. Egy név nélküli névteren belül deklarált változók és függvények hasonlóan viselkednek, mintha eléjük lenne írva a `static` kulcsszó.

```
namespace
{
    int x;
    std::string y;
    void f() {}
}
```

10.4.1. Megjegyzés. A `static` osztályon belüli jelentéséről később lesz szó.

11. Függvény túlterhelés

Térjünk vissza a korábban megírt swap függvényünkhöz.

```
void swap(int &a, int &b)
{
    int tmp = a;
    a = b;
    b = tmp;
}
```

Ez a függvény addig jó, amíg csak `int`-eket szeretnénk megcserélni. Mi van, ha `std::string`-eket kéne? A megoldás egyszerű, **túlterheljük** (*overload*) a `swap` függvényt.

```
void swap(std::string &a, std::string &b)
{
    std::string tmp = a;
    a = b;
    b = tmp;
}
```

Túlterhelésnek azt nevezzük, amikor két vagy több függvénynek a neve azonos, de a paramétereik különböznek. Tagfüggvényeket konstansság alapján is túl lehet terhelni.

11.0.1. Megjegyzés. A később elhangzó osztályok tagfüggvényeinél a függvény konstanssága is számít (azonos nevű és paraméter listájú függvény különöző konstansséggal ugyanúgy túlterhelésnek számít).

11.1. Operátor túlterhelés

Ahogy láttuk a 2. gyakorlat elején, lehetőségünk van függvényeket túlterhelni. Ez operátorokra is igaz. Ha példaként vesszük a lineáris algebrából tanult rendezett valós számhármasokat ($\mathbb{R}^3$), lehetőségünk van arra, hogy a tanultak alapján definiáljuk a köztük értelmezett összeadást.

```
struct LinAlgVector
{
    double x1, x2, x3;
};

LinAlgVector operator+(const LinAlgVector &lhs, const LinAlgVector &rhs)
{
    LinAlgVector ret;
    ret.x1 = lhs.x1 + rhs.x1;
    ret.x2 = lhs.x2 + rhs.x2;
    ret.x3 = lhs.x3 + rhs.x3;
    return ret;
}
```

```
int main()
{
    LinAlgVector a, b;
    a.x1 = 1; a.x2 = 2; a.x3 = 3;
    b.x1 = 1; b.x2 = 1; b.x3 = 1;

    LinAlgVector c = a + b;
}
```

A `main` függvényben lévő értékadás ezzel ekvivalens: `c = operator+(a, b)`, így láthatjuk, hogy az operátorok túlterhelése gyakorlatilag a függvénytúlterhelés speciális esete.

Írjuk meg a `print` függvényt 3D vektorokra! A gyakran kiíratáshoz használt jobb shift operátor (*right shift operator*), a `<<` is túlterhelhető. Mivel mi az `std::cout` változóval szeretnénk majd kiíratni, melynek típusa `std::ostream`, így a függvényünk első paramétere egy ilyen típus lesz, a második meg egy `LinAlgVector` típus.

```
/* ... */

std::ostream& operator<<(std::ostream& os, const LinAlgVector &l)
{
    os << l.x1 << ' ' << l.x2 << ' ' << l.x3;
    return os;
}

int main()
{
    /* ... */
    std::cout << c << std::endl; // 2 3 4
}
```

Feltűnhet, hogy a stream objektumra mutató referenciát a függvény végén vissza is adjuk, hogy tudjuk a kiíratást láncolni.

12. Pointer aritmetika

12.1. Konstans korrektség

Térjünk vissza a mutatókhoz. Volt már szó konstans mutatókról, ám konstans**ra** mutató mutatókról még nem.

```
const int ci = 6;
int *p = &ci;
```

A fenti kód nem fordul le, mert `ci` konstans, de `p` nem egy nem konstansra mutató pointer. Ez sértené a C++ban ismert **konstans korrektséget** (const correctness). A probléma forrása az, ha fenti értékadás lefordulna, akkor `ci` értékét tudnánk módosítani `p`-n keresztül.

Konstans korrektség: ha egy értéket konstansnak jelölünk, azt nem módosíthatjuk a program futása során.

```
const int ci = 6;
const int *p = &ci;
```

Ez már lefordul, hiszen a `p` egy konstansra mutató pointer, azaz mutathat konstans változókra. Egy konstansra mutató pointer **nem tudja megváltoztatni** a mutatott értéket. Viszont egy konstansra mutató mutatót át lehet állítani egy másik memóriacímre.

```
const int ci = 6;
const int *p = &ci;

int c = 5;
p = &c;
```

A fenti kód is szabályos, konstansra mutató pointerrel nem konstans is értékre mutathatunk. Érdemes átgondolni ennek a következményeit, hisz `c` nem konstans, ezért az értékét továbbra is módosíthatjuk (csak nem `p`-n keresztül)! Egy konstansra mutató mutató nem azt jelenti, hogy a mutatott érték sosem változhat meg. Csupán annyit jelent, hogy a mutatott értéket ezen a mutatón keresztül nem lehet megváltoztatni.

```
const int *p = &ci;
int c = 5;
p = &c;
c = 5;
```

A `const` kulcsszó több helyre is kerülhet.

```
const int *p;
int const *p;
```

A fenti két sor ugyanazt jelenti, a mutatott értéket nem lehet megváltoztatni a mutatón keresztül.

```
int * const p;
const int * const p;
int const * const p;
```

Amennyiben a * után van a `const`, akkor egy **konstans pointert kapunk**, mely megváltoztathatja a mutatott értéket, de nem mutathat másra (konstans pointer $\neq$ konstanra mutató pointer). Egy létezik konstansra mutató konstans mutató is, amin keresztül nem lehet megváltoztatni a mutatott értéket és a mutatót sem lehet máshova átállítani.

12.2. Mutatóra mutató mutatók

Mutatóra mutató mutatók is léteznek. Néhány példa:

```
int *p;
int **q = &p;
int ***r = &q;
```

Példaképp `q`-n keresztül meg tudjuk változni, `p` hova mutasson.

```
int c, d;
int *p = &c;
int **q = &p;
*q = &d;
```

A megfelelő szinten a mutatók konstansá tételével még bonyolultabb példákat kaphatunk:

```
int c, d;
int *p = &c;
int * const *q = &p;
*q = &d; // forditasi hiba
```

Mivel `q` egy `int`-re mutató konstans mutatóra mutató mutató, így csak egy olyan mutatóval tudunk rámutatni, ami egy `int`-re mutató konstans mutatóra mutató mutatóra mutató mutató.

```
int c, d;
int *p = &c;
int * const *q = &p;
int *const ** const r = &q;
```

12.2.1. Megjegyzés. Megnyugtatás végett, ritkán van szükség mutatóra mutató mutatónál bonyolultabb szerkezetre.

12.3. Függvény pointerek

C++ban lehetőségünk van arra is, hogy függvényeket adjunk át paraméterként.

```
int add(int a, int b)
{
    return a + b;
}

int mul(int a, int b)
{
```

```
    return a * b;
}

int reduce(int *start, int size, int initial, int (*op)(int, int))
{
    int ret = initial;
    for (int i = 0; i < size; i++)
    {
        ret = (*op)(ret, start[i]);
    }
    return ret;
}

int main()
{
    int t[] = {1,2,3,4,5};
    std::cout << reduce(t,5,0,&add) << std::endl;
    std::cout << reduce(t,5,0,&mul) << std::endl;
}
```

Itt `reduce` egy olyan paramétert is vár, mely igazából egy függvény, amely `int`-et ad vissza, és két `int`-et vár paraméterül.

12.3.1. Megjegyzés. A szavakba öntés segíthet a megértésben: `op` egy olyan függvényre mutató mutató, melynek két `int` paramétere van, és `int` a visszatérési értéke.

A kódban feltűnhet, hogy a tömb mellé paraméterben elkértük annak méretét is. Ennek az az oka, hogy a `t` tömb egy `int`-re mutató mutatóvá fog konvertálódni a paraméter átadás során, ami a tömb első elemére mutat. Ennek hatására elvesztjük azt az információt, hogy mekkora volt a tömb (a tömbök és paraméterátadás kapcsolatról később bővebben lesz szó). Így át kell adni ezt az információt is.
Mellékesen, egy függvény átadásakor csak függvénypointert tudunk átadni. Egy függvénypointeren a függvény meghívása az egyetlen értelmes művelet. Így a `&` jel elhagyható függvényhíváskor és az `op` elől is elhagyható a * a paramétereknél.

```
int reduce(int *start, int size, int initial, int op(int, int)))
{
    //...
}

int main()
{
    int t[] = {1,2,3,4,5};
    std::cout << reduce(t,5,0,add) << std::endl;
    std::cout << reduce(t,5,0,mul) << std::endl;
}
```

13. Tömbök

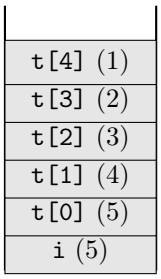
A tömb a C++ egy beépített adatszerkezete, mellyel tömb azonos típusú elemet tárolhatunk és kezelhetünk egységesen. Előzménytárgyakból már megismertük valamennyi funkcionalitását, ám számos veszélyét még nem.

```
int main()
{
    int i = 5;
    int t[] = {5,4,3,2,1};
}
```

`t` egy 5 elemű **tömb**. Nézzük meg, mekkora a mérete (figyelem, ez **implementációfüggő!**)!

```
std::cout << sizeof(i) << std::endl;
std::cout << sizeof(t) << std::endl;
```

A `sizeof` operátor megadja a paraméterként megadott típus, vagy objektum esetében annak típusának méretét (bővebben később). Ez minden implementációra specifikus. Azt látjuk, hogy mindig ötszöröse lesz a `t` az `i`-nek. Azaz a tömbök tiszta adatok. Stacken ábrázolva így képzeljük el:



A `main` függvény változói.

13.1. Biztonsági rések nem definiált viselkedés kihasználásával

Irassuk ki a a tömb elemeit! A példa kedvéért rontsuk el a kódot.

```
for (int i = 0; i < 6; i++) // Hupsz. A csak 5 elem van.
{
    std::cout << t[i] << std::endl;
}
```

Itt látható, hogy túl fogunk indexelni. Ez nem definiált viselkedéshez vezet. Várhatóan memóriaszemetet fog kiolvasni az utolsó elem helyett, de sose tudhatjuk pontosan mi fog történni. Fordítási időben ezt a hibát a fordító nem veszi észre. A gyakorlaton a programot futtatva nem következett be futási idejű hiba.
Most növeljük meg az elemeket, és indexeljünk túl egészen 100ig!

```
for (int i = 0; i < 100; i++)
{
    ++t[i];
}
std::cout << "sajt" << std::endl;
```

Ez a program továbbra is nem definiált viselkedést tartalmaz. Mivel több memóriához nyúlunk hozzá indokolatlanul, ezért nagyobb rá az esély, hogy futási idejű hibába ütközzünk. Az órán a sajt szöveg ki lett írva, mégis kaptunk egy szegmentálási hibát (*segmentation fault*).

```
for (int i = 0; i < 100000; i++)
{
    ++t[i];
}
std::cout << "sajt" << std::endl;
```

A túlindexelést tovább fokozva a program még mielőtt sajt-ot ki tudta volna írni, szegmentálási hibával leállt. Ez jól demonstrálja, hogy ugyanolyan jellegű a hibát követtük el, de mégis más volt a végeredmény. Ez az egyik ok, amiért veszélyesek a nem definiált viselkedések. Mivel számos különböző hibát okozhatnak, ezért a diagnosztizálásuk sem mindig egyszerű. Az alábbi kód szemléltet egy példát, hogyan lehet biztonsági rés a nem definiált viselkedésből.

```
#include <iostream>
#include <string>

int main()
{
    int t[] = {5,4,3,2,1};
    int isAdmin = 0;
    std:string name;
    std::cin >> name;
    for (int i = 0; i < name.size(); ++i)
```

```
    {
        t[i] = 1;
    }
    if (name == "pityu")
        isAdmin = 1;
    std::cout << "Admin?: " << (isAdmin != 0 ) << std::endl;
}
```

Ha a programnak `pityu`-t adunk meg amikor be akarja olvasni `name`-et, akkor minden rendben. De mivel a forráskódot ismerjük, azért ha hosszú nevet adnánk (nagyobb mint 5), akkor a túlindexelés miatt ki tudjuk használni a nem definiált viselkedéseket. Az is előfordulhat, hogy az `isAdmin` memóriacímére írunk, és elérjük, hogy a szoftver adminként authentikáljon valakit, aki nem az.

Hogyan lehet ezeket a hibákat elkerülni? Túl azon, hogy figyelni kell, vannak programok amik segítenek. Ehhez használhatunk `sanitizer`-eket. Ezek módosítanak a fordító által generált kódon. Létrehoz ellenőrzéseket, amik azelőtt észrevesznek bizonyos nem definiált viselkedéseket, mielőtt azok megtörténnének. Pl. itt a túlindexelés, egy futási idejű hibához és egy jól olvasható hibaüzenethez vezetne. Használatukhoz elég egy extra paranccsal fordítanunk:

```
g++ main.cpp -fsanitize=address
```

A sanitizerek csa abban az esetben találnak meg egy hibát, ha a probléma előfordul (azaz futási időben, nem fordítási időben ellenőriz). Amennyiben előfordul, akkor elég pontos leírást tudunk kapni arról, hogy merre van a probléma. Fordítási időben a figyelmeztetések használata segíthet bizonyos hibák elkerülésében.

```
g++ main.cpp -Wall -Wextra
```

A fenti két kapcsoló szintén extra ellenőrzéseket vezet be, de nem változtatják meg a generált kódot.

13.2. Hivatkozás tömb elemeire

```
#include <iostream>

int main()
{
    int t[] = {5,4,3,2,1};
    int *p = t;
    std::cout << *p << std::endl; // 5
    std::cout << sizeof(int) << std::endl; // implementációfüggő, legyen x
    std::cout << sizeof(p) << std::endl; // implementációfüggő, legyen y
    std::cout << sizeof(t) << std::endl; // 5*x
}
```

Könnyű azt hinni (hibásan), hogy a pointerek ekvivalensek a tömbökkel. A fenti program jól szemlélteti, hogy ez nem igaz. A tömb típusa tartalmazza azt az információt, hogy hány elemű a tömb. Egy pointer típusa csak azt az információt tartalmazza, hogy a mutatott elem mekkora. Számos más különbség is van. A tévhit oka az, hogy tömb könnyen konvertálódik első elemre mutató pointerré.
Egy tömb adott elemére több módon is hivatkozhatunk:

```
*(p + 3) == *(3 + p) == p[3] == 3[p]
```

```
#include <iostream>

int main()
{
    int t[][3] = {{1,2,3},{4,5,6}};
    return 0;
}
```

Tekintsük a fenti két dimenziós tömböt. Az első `[]` jelek közt nincs méret, mert a fordító az inicializáció alapján meg tudja állapítani. A második dimenzió méretének megadása viszont kötelező.

Fentebb a tömböknél megadott ekvivalenciát a mátrixra alkalmazva számos indexelési módot le tudunk vezetni:

```
t[1][] == *(*(t+1)+0) == *(1[t]+0) == 0[1[t]] == 0[*(t+1)] == *(t+1)[0] == 1[t][0]
```

13.2.1. Megjegyzés. Ahhoz, hogy egy olyan függvényt írjunk, ami minden méretű tömböt elfogad paraméterül, a legegyszerűbb megoldás, ha hagyjuk, hogy a tömb átkonvertálódjon egy első elemre mutató pointerré, és átadjuk külön paraméterben a tömb méretét. Bár van megoldás arra is, hogy egy darab "rugalmas" függvényt írjunk, és az egész tömbről csak egy paramétert vegyünk át. Majd a 7-8. gyakorlaton lesz részletesen szó, de a következő a szintaxis:

```
template <class T, int ArraySize>
void ( T (&param)[ArraySize] )
{
    //...
}
```

A működésének az elvét még nem baj, ha nem értjük. A háttérben egy template paraméter dedukció fog végbemenni: a fordító kitalálja `param` méretét.

A tömbök átvétele paraméterként azért ilyen körülményes, mert egy tömbnek a méretét fordítási időben ismernünk kell. Ha változó méretű tömböt várnánk paraméterül, az szembemenne ezzel a követelménnyel.

13.2.2. Megjegyzés. Előzmény tárgyakból elképzelhető, hogy azt tanultuk, hogy egy tömb méretének egy változót is megadhatunk. Ezt a `gcc` fordító elfogadja és jó kódot is generál belőle. De ez egy nem szabványos kiterjesztés, ezért nem garantált, hogy ezt minden fordító megteszi. Ez jól demonstrálja, hogy a fordítók nem mindenben követik szorosan a szabványt.

13.3. Tömbök átadása függvényparaméterként

Próbáljunk meg egy tömböt érték szerint átadni egy függvénynek!

```
#include <iostream>

void f(int t[])
{
    std::cout << sizeof(t) << std::endl;
}

int main()
{
    int t[] = {1,2,3,4,5};
    std::cout << sizeof(t) << std::endl;
    f(t);
}
```

Kimenet: `20 8` (implementáció függő)
Bár azt hihettük, hogy `t` tömb méretét írattuk ki ket alkalommal, valójában amikor azt érték szerint próbálunk meg átadni egy tömböt, az átkonvertálódik a tömb elejére mutató pointerré.

```
void f(int t[8])
{
    std::cout << sizeof(t) << std::endl;
}
```

Hiába adunk meg egy méretet a tömbnek a függvény fejlécében, még mindig egy pointer mérete lesz a második kiírt szám. Az a tanulság, hogy ha érték szerint akarunk átadni egy tömböt, az át fog konvertálódni pointerré. A legszebb az lenne, ha a fenti szintaxis nem fordulna le. Ennek azonban történelmi oka van, a C-vel való visszafelé kompatibilitás miatt fordul le.

13.3.1. Megjegyzés. Tömböt értékül adni a szabvány szerint nem is lehet: `int *t2[5] = t` nem helyes.

Korábban megismerkedtünk egy módszerrel, mely segítségével egy tömb méretét (elemszámát) paraméterátadás után is megőriztük:

```
#include <iostream>

void f(int *t, int size) // új paraméter!
{
```

```
    std::cout << sizeof(t) << std::endl;
}

int main()
{
    int t[] = {1,2,3,4,5};
    std::cout << sizeof(t) << std::endl;
    f(t, sizeof(t)/sizeof(t[0]));
}
```

13.3.2. Megjegyzés. Amennyiben C++11ben programozunk, érdemes az `std::array`-t használnunk, ami olyan, mint egy tömb, de nem tud pointerré konvertálódni és mindig tudja a méretét.

Ha szeretnénk egy tömböt egy darab paraméterként átadni, megpróbálhatunk egy tömbre mutató pointert létrehozni. Azonban figyelni kell a szintaktikára, ha `int *t[5]`-t írunk, egy öt elemű intre mutató pointereket tároló tömböt kapunk.

Ha tömbre mutató mutatót szeretnék, így csinálhatjuk:

```
void g(int (*t)[5])
{
    std::cout << sizeof(t) << std::endl;
}
```

Azonban ez még mindig egy pointer méretét fogja kiírni, mert a `t` az egy sima mutató! Ahhoz, hogy megkapjuk, mire mutat, dereferálnunk kell, így a `sizeof` paraméterének `*t`-t kell megadni, ha a tömb méretére vagyunk kíváncsiak.

13.3.3. Megjegyzés. Ha refenreciával vennénk át `t`-t, az is hasonlóan nézne ki: `int (&t)[5]`.

Ha eltérő méretű tömböt próbálunk meg átadni, akkor nem fordul le a kód, mert nem egy 5 elemű tömbre mutató mutató 6 elemű tömbre mutató mutatóvá konvertálódni.

```
int main()
{
    int a[6];
    g(&a); //forditasi hiba!
    int b[5];
    g(&b); //ok
}
```

14. Literálok

14.1. Karakterláncok

Mi lesz a `"Hello"` karakterlánc literál típusa?

Egy konstans karakterekből álló 6 méretű tömb (`const char[6]`). Azért 6 elemű, mert a karakterlánc literál végén el van tárolva a végét jelző `\0` karaktert.

H	E	L	L	O	\0

```
int main()
{
    char* hello= "Hello";
    hello[1] = 'o';
}
```

A fenti kódban megsértettük a konstans korrektséget, hisz egy nem konstansra mutató pointerrel mutatuk egy konstans karakterlánc literál első elemére. Ennek ellenére, a fenti kód lefordul. Ennek az az oka, hogy az eredeti C-ben nem volt const kulcsszó, a kompatibilitás végett ezért C++ban lehet konstans karakterlánc literál elemire nem konstansra mutató pointerrel mutatni.

14.1.1. Megjegyzés. Ezt a fajta kompatibilitás miatt meghagyott viselkedést kerülni kell. Lefordul, de kapunk rá warningot.

Ha módosítani próbáljuk a karakterlánc literál értékét, az nem definiált viselkedéshez vezet.
Futtatáskor linuxon futási idejű hibát kapunk, méghozzá szegmentálási hibát. Ennek az az oka, hogy a konstansok értékei readonly memóriában vannak tárolva, aminek a módosítását nem engedi az operációs rendszer.

Ez jól rámutat arra, hogy miért is nem jó az, ha a fenti konverziót megengedjük.

14.2. Szám literálok

Függően attól, hogy egy szám literált hogyan írunk C++ban, mást jelenthet:

`5`	`int`
`5.`	`double`
`5.f`	`float`
`5e-4`	`double`, értéke 0.0005
`5e-4f`	`float`
`0xFF`	16-os számrendszerben ábrázolt `int`
`012`	8-as számrendszerben ábrázolt `int`
`5l`	`long int`
`5u`	`unsigned int`
`5ul`	`unsigned long int`

14.2.1. Megjegyzés. Alapértelmezetten minden `int` egy `signed int`.

14.2.2. Megjegyzés. Viszonylag kevés esetben éri meg `float`-ot használni `double` helyett. Modern CPU-k ugyanolyan hatékonyan dolgoznak mind a kettővel, így érdemesebb a pontosabbat választani. (Ha magát a GPU-t programozzuk, az lehet egy kivétel.)

Létezik C++ban `signed` kulcsszó, mely a char miatt lett bevezetve. A char is egész számokat tartalmaz, de az implementáció függő, hogy a `char` singed vagy unsigned értéket tartalmaz-e.

14.2.3. Megjegyzés. Érdemes mindig `int`et használnunk, hanincs jó okunk arra, hogy mást használjunk. Az `int`-el általában a leghatékonyabb a processzor.

A `sizeof(char)` mindig 1-et ad vissza. A karakter mérete mindig az egység. Minden más típusra a `sizeof` függvény azt adja vissza, hogy paraméterül megadott objektum vagy típus mérete hányszorosa a `char`nak.
Attól, hogy `sizeof(char) == 1`, a `char` mérete byteokban még implementáció függő.
A lebegőpontos számok mindig rendelkeznek előjellel.

A `char` méretén túl minden másnak a mérete implementációfüggő, bár a szabvány kimond pár relációt:

$$\texttt{sizeof(X)} == \texttt{sizeof(signed X)} == \texttt{sizeof(unsigned X)}$$
$$\texttt{sizeof(float)} \leq \texttt{sizeof(double)} \leq \texttt{sizeof(long double)}$$
$$\texttt{sizeof(short)} \leq \texttt{sizeof(int)} \leq \texttt{sizeof(long)}$$
$$\texttt{sizeof(char)} \leq \texttt{sizeof(bool)}$$

15. Struktúrák mérete

```
#include <iostream>

struct Hallgato
{
    double atlag;
```

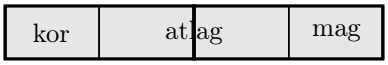
```
    int kor;
    int magassag;
}

int main()
{
    std::cout << sizeof(double) << std::endl;
    std::cout << sizeof(int) << std::endl;
    std::cout << sizeof(Hallgato) << std::endl;
}
```

A gyakorlaton használt gépen a `double` mérete 8, az `int` mérete 4, `Hallgato`-é 16. Ezen azt látjuk, hogy a `Hallgato` tiszta adat.

```
struct Hallgato
{
    int kor;
    double atlag;
    int magassag;
}
```

Miután átrendeztük a mezők sorrendjét, és újra kiírjuk a struktúra méretét, akkor a válasz 24. Ennek az oka az, hogy míg az első esetben így volt eltárolva a memóriában: (ne feledjük, ez még mindig implementációfüggő!)

Azaz, `atlag`, illetve `kor` és `magassag` pont efértek 1-1 gépi szóban. Viszont, ha megcseréljük a sorrendet, ez már nem lesz igaz:



Itt az `atlag` két fele két különböző gépi szóba kerülne. Ez a ma használt processzorok számára nem hatékony, hiszen az átlag értékének kiolvasásához vagy módosításához két gépi szót is olvasni vagy módosítani kéne (a legtöbb processzor csak szóhatárról tud hatékonyan olvasni).

A fenti elrendezés hatékonyabb, bár 3 gépi szót használ. Ebben az esetben a fordító *padding*et illeszt be a mező után. Ennek hatására hatékonyan olvasható és módosítható minden mező. Cserébe több memóriát foglal a struktúra.

A szabvány kimondja, hogy egy `struct` mérete az adottagok méreteinek összegénél nagyobb vagy egyenlő.

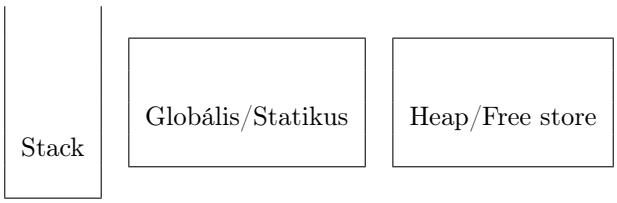
Az, hogy egy gépi szó mekkora, implementációfüggő.

Egy `struct` egyes adattagjaira a pont operátor segítségével hivatkozhatunk:

```
int main()
{
    //...
    Hallgato a;
    std::cout << a.kor << std::endl;
    Hallgato b = a;
    b.magassag = 3;
}
```

16. A C++ memóriamodellje

A C++ szabvány több memóriatípust különít el. Ezek közül elsősorban a stack-et használtuk eddig.



16.1. Stack

A második gyakorlaton már volt szó részletesebben a stack működéséről. Jusson eszünkbe egy nagyon fontos tulajdonsága: a blockok végén a változók automatikusan megsemmisülnek. Ebben az esetben nem a programozó feladata a memória felszabadítása.

A stack-en létrehozott változókat szokás **automatikus változók**nak (*automatic variable*) is hívni.

Tekintsük a második gyakorlatról már ismerős kódrészletet.

```
#include <iostream>

int f()
{
    int x = 0;
    ++x;
    return x;
}

int main()
{
  for(int i = 0; i < 5; ++i)
      std::cout << f() << std::endl;
}
```

Kimenet: `1 1 1 1 1`

A stacken lérehozott változók kezelése nagyon kényelmes, mert jól látható, mikor jönnek jönnek létre, mikor semmisülnek meg. Azonban előfordulhat, hogy nem szeretnénk, hogy az `x` változó élettartama megszűnjön a block végén. Ilyenkor egy lehetőség a statikus változók használata.

16.2. Globális/statikus tárhely

Írjuk át a fenti `f` függvényt, hogy `x` ne automatikus, hanem **statikus változó** (*static variable*) legyen!

```
int f()
{
    static int x = 0;
    ++x;
    return x;
}

int main() {/*...*/}
```

Kimenet: `1 2 3 4 5`
Ebben az esetben azonban a függvény első hivásától a program futásának végéig benne marad a memóriában az `x`, így mindig egyre nagyobb számokat ad majd `f()` vissza. Az `x` inicializációja egyszer történik meg, a függvény első hívásakor.

16.2.1. Megjegyzés. Nem szeretjük a `static` változókat. Például a több szálú programok esetén különösen kerülendő ez a programozási stílus.

A globális változók és a statikus változók a memória ugyanazon területén jönnek létre. Ezért hívjuk ezt a területet globális/statikus tárhelynek.

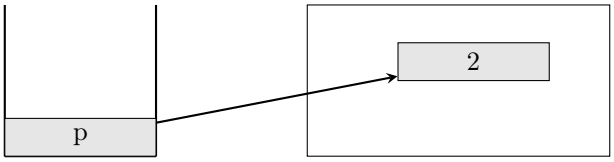
Amennyiben azt szeretnénk, hogy `x` ne semmisüljön meg a block végén, de ne is maradjon a program futásának a végéig a memóriában, arra is van lehetőség. Ebben az esetben viszont a programozó felel a memória kezeléséért.

16.3. Heap/Free store

A heapen létrehzoott változókat **dinamikus változók**nak (*dynamic variable*) is szokás szokás hívni. A heap segítségével nagy szabadságra tehetünk szert, azonban ez a szabadság nagy felelősséggel is jár.

```
int main()
{
    int *p = new int(2);
    delete p;
}
```

Fentebb láthatjuk hogyan lehet egy intnek lefoglalni helyet a heapen. Fontos, hogy a stack-et nem kerültük meg, mert szükségünk van egy pointerre, mely a heap-en lefoglalt címre mutat (`p`).
A mutató által mutatott területet a `delete` operátorral tudjuk felszabadítani.



A heapen nincs a lefoglalt területnek nevük, így mindig szükségünk lesz egy mutatóra, hogy tudjunk rá hivatkozni. **Ha egyszer lefoglalunk valamit a heap-en, gondoskodni kell arról, hogy felszabadítsuk.** Az egyik leggyakoribb hiba a dinamikus memóriakezelésnél, ha a memóriát nem szabadítjuk fel. Ilyenkor a lefoglalt memóriaterületre hivatkozni már nem tudunk de lefoglalva marad: elszivárog (*memory leak*).

Bár az operációs rendszer megpróbál minden, a program által lefoglalt memóriát felszabadítani a futás befejeztével, de nem mindenható. Előfordulhat, hogy egyes platformokon újraindításig nem szabadul fel a memória. Emellett ameddig a program fut, több memóriát fog használni, mint amennyire szüksége van. Ez növelheti a szerverpark költségeit vagy ronthatja a felhasználói élményt.

A dinamikusan lefoglalt memória szabályos felszabadítását számos dolog nehezíti. Fényes példa erre a kivételkezelés, melynél hamarabb megszakadhat a függvény végrehajtása mintsem, hogy felszabadítson minden memóriát. Előfordulhat, hogy egy memóriaterületet kétszer szabadítunk fel, ami nem definiált viselkedés.

Előfordulhat, hogy egy már felszabadított memóriaterületre akarunk írni vagy onnan olvasni. Sajnos ilyen jellegű hibát könnyű véteni, hisz a `delete` a `p` által mutatott memóriaterületet, nem a `p`-t fogja törölni. A `p` továbbra is használható.

16.3.1. Megjegyzés. A nullpointer törlésekor nem történik semmi (*no-op*).

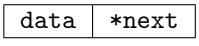
16.3.2. Megjegyzés. Amint elvesztettük az utolsó mutatót, ami egy adott lefoglalt memóriacímre mutat, az garantáltan elszivárgott memória. A szabvány nem foglal magában semmilyen lehetőséget ezeknek a visszaszerzésére.

Láthatjuk, hogy a heap használata hibalehetőségekkel teli, ráadásul az allokálás (memória lefoglalás) még lassabb is, mintha a stack-et használnánk. De miért használjuk mégis? Ha meg lehet oldani, hogy a stack-en tudjunk tárolni valamit, tegyük azt. A stacken azonban véges, hamar be tud telni (*stack overflow*), illetve kötött a változók élettartama. A heap-en e téren sokkal nagyobb a szabadságunk.

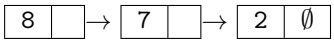
17. Osztályok felépítése

A következő pár gyakorlaton egy láncolt listát fogunk implementálni, mely jól demonstrálja majd a dinamikus memóriakezelés veszélyeit is.

A láncolt lista egy olyan konténer, melynek minden eleme egy olyan listaelem, mely tartalmaz egy mutatót és (legalább egy) adatot tároló objektumot. A listaelem mutatója rámutat a lista következő elemére, és az utolsó elem pointere pedig egy nullpointer.



Egy listaelem.



3 elemű láncolt lista.

17.1. Struct-ok

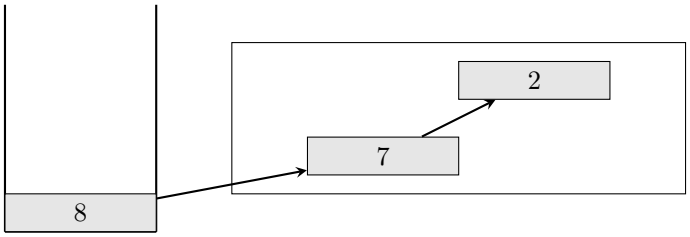
Egy láncolt lista elemét implementálhatjuk pl. így:

```
struct List
{
    int data;
    List *next;
};
```

Alkalmazzuk is ezt úgy, hogy a listaelemek dinamikusan legyen eltárolva!

```
int main()
{
    List *head = new List;
    head->data = 8; //(*head).data    ==    head->data
    head->next = new List;

    head->next->data = 7;
    head->next->next = new List;

    head->next->next->data = 2;
    head->next->next->next = NULL;

    delete head;
    delete head->next;
    delete head->next->next;
}
```

Ezen a ponton készen vagyunk, hisz `List` használható láncolt listaként (bár valójában igen kényelmetlen).



Sajnos a törlést rossz: először törljük a fejelemet (mely az első elemre mutat), viszont az első elem segítségével tudnánk a többi elemet elérni, így mikor a második listaelemet törölnénk, `head` már egy felszabadított memóriaterületre mutat. Ezt törlés utáni használatnak (*use after delete*) szokás nevezni és nem definiált viselkedés. A megoldás:

```
delete head->next->next;
delete head->next;
delete head;
```

17.1.1. Megjegyzés. A heap-en arra is figyelni kell, hogy jó sorrendben szabadítsuk fel a memóriát. Ha rossz sorrendben szabadítjuk fel az objektumokat, könnyen a fentihez hasonló hibát vagy memória szívárgást okozhatunk.

A változók a stacken a létrehozás sorrendjéhez képest fordított sorrendben semmisülnek meg, pont emiatt.

Ez a „láncolt lista” eddig elég szegényes. A fő gond az, hogy nagyon sokat kell írni a használatához. Ez sért egy programozási elvet, a DRY-t: *Dont Repeat Yourself*. Itt sokszor írjuk le közel ugyanazt – erre kell, hogy legyen egy egyszerűbb megoldás. Írjunk függvényt az új listaelem létrehozásához!

```
List *add(List *head, int data)
{
    if (head == 0)
    {
```

```
        List *ret = new List;
        ret->data = data;
        ret->next = 0;
        return ret;
    }
    head->next = add(head->next, data);
    return head;
}
```

Ez egy olyan rekurzív függvény, mely addig hívja saját magát, míg a paraméterként kapott lista végére nem ér (azaz a `head` egy nullpointer). Amikor oda elér, létrehoz egy új listaelemet és azt visszaadja. A rekurzió felszálló ágában a lista megfelelő elemeit összekapcsolja.

Írjunk egy függvényt a lista által birtokolt memória felszabadítására is.

```
void free(List *head)
{
    if (head == 0)
        return;
    free(head->next);
    delete head;
}
```

Itt a rekurzió szintén a lista végéig megy. A rekurzió felszálló ágában történik a listaelemek felszabadítása. Ennek az oka, hogy a felszabadítás a megfelelő sorrendben történjen meg.

17.1.2. Megjegyzés. A rekurzív függvények nem olyan hatékonyak, mint az iterativ (pl. `for` vagy `while` ciklus) társaik. Továbbá a sok függvényhívás könnyen stack overflow-hoz vezetnek. Azonban jó agytornák, és segíthetnek az alapötletben. Egy rekurzív függvényt mindig át lehet írni iteratívvá.

Beszéljünk arról, mennyi a teher a felhasználón. Eddig tudnia kellett, milyen sorrendben kell felszabadítani az elemeket a listán belüle, de most már elég arra figylenie, hog lista használata után meghívja a `free` függvényt. A felhasználó így kisebb eséllyel követ el hibát, több energiája marad arra, hogy az előtte álló problémát megoldja. Legyenek a függvényeink és osztályaink olyanok, hogy **könnyű legyen őket jól használni, és nehéz legyen rosszul**.

17.2. Osztályra statikus változók

Teszteljünk!

```
int main()
{
    List *head = 0;
    head = add(head, 8);
    head = add(head, 7);
    head = add(head, 2);

    free(head);
}
```

A program lefordult, és a gyakorlaton tökéletesen le is futott. Azonban, ha történt memory leak vagy double free, esetleg use after free, az nem definiált viselkedés. Ezért nem lehetünk benne biztosak, hogy valóban nem történt memóriakezeléssel kapcsolatos hiba. A sanitizerek segítségével megyőződhetünk róla, hogy nem követtünk el ilyen jellegű hibát.

Az osztályon belül statikusként deklarált változókat osztályszintű változóknak is hívjuk, ugyanis minden, az osztályhoz tartozó objektum ugyanazon a statikus változón „osztozkodik”. Ha az egyiken keresztül azt a változót módosítjuk, a többiben módosulni fog. Élettartamuk és láthatóságuk a program elejétől végéig tart.
Hozzunk létre `List`-ben egy számlálót, ami számon tartja mennyi objektumot hoztunk belőle létre, és semmisítettünk meg! Ezzek a trükkel megnézhetjük, hogy elfelejtettünk-e felszabadítani listaelemet.

```
struct List
{
    int data;
```

```
    List *next;

    static int count; // !
};

int List::count = 0;

List *add(List *head, int data)
{
    if (head == 0)
    {
        List *ret = new List;
        List::count++; // !
        ret->data = data;
        ret->next = 0;
        return ret;
    }
    head->next = add(head->next, data);
    return head;
}

void free(List *head)
{
    if (head == 0)
    return;
    free(head->next);
    List::count--; // !
    delete head;
}

int main()
{
    List *head = 0;
    head = add(head, 8);
    head = add(head, 7);
    head = add(head, 2);

    free(head);
    std::cout << List::count; // !
}
```

Osztályszintű változókat csak osztályon kívül tudunk definiálni (ezek alól kivételt képeznek az osztályszintű konstans változók), ezért látható az osztály után a következő sor:

```
int List::count = 0;
```

Ezzel a kis módosítással meg is kapjuk a kívánt kimenetet: `0`. Ez alapján tudhatjuk, hogy minden objektum törlésre került.

17.2.1. Megjegyzés. A fenti módosítások csak gyakorlás célját képezték, az elkészítedő listának nem része a számláló.

Ha azonban egy elemet kétszer töröltünk, egyet meg elszivárogtattunk, az nem feltétlen nem derül ki. Ilyenkos a sanitizerek segíthetnek:

```
g++ list.cpp -fsanitize=address -g
```

A sanitizerekről bővebben lásd a 3. gyakorlat anyagát. Határozott előrelépést értünk el, de van még hova fejleszteni a listánkat. Szerencsére nem csak adattagokat, de tagfüggvényeket is tudunk struct-okba írni.

17.3. Konstruktorok

Egy trükkel megoldható, hogy tömörebb szintaxissal tudjuk inicializálni a listaelemeinket.

```
struct List
{
    List(int _data, List *_next = 0) : data(_data), next(_next) {}

    int data;
    List *next;
};
```

A fenti tagfüggvényt, vagy metódust **konstruktor**nak (*constructor*, vagy röviden *ctor*) hívjuk. A konstruktorok hozzák létre az objektumokat; vannak paraméterei, és nincs visszatérési értéke. A fenti konrtruktor még egy alapértelmezett paraméterrel is rendelkezik - ha mi csak egy `int` paraméterrel hívjuk meg a konstruktort, akkor a `_next`-et alapértelmezetten nullpointernek veszi.

Azonban a struktúránk működött eddig is, pedig nem írtunk konstruktort. Ha konstruktorra szükség van objektum létrehozáshoz, akkor hogyan lehet ez? Úgy, hogy a fordító a hiányzó kulcsfontosságú függvényeket legenerálja nekünk. Létrehoz (többek között) egy un. **default konstruktort**, ha mi explicit nem hoztunk létre konstruktort. A default konstruktor 0 paraméterrel meghívható. Fontos azonban, ha mi írunk egy konstruktort, akkor a fordító már nem fog generálni ilyet.

Így a következőféleképpen tudunk egy `List` típusú objektumot létrehozni:

```
List head(5); //ok, létrehoz egy 5 értékkel rendelkező, 1 elemű listát
List head2; //nem ok, már nincs paraméter nélküli konstruktor
```

A konstruktor fejléce után található egy un. **inicializációs lista**. Az inicializációs listával rendelkező konstruktor hasonló jelentéssel bír, mint a következő kód:

```
List(int _data, List *_next = 0)
{
    data = _data;
    next = _next;
}
```

Az inicializációs lista használata azonban hatékonyabb. Mire a konstruktor törzséhez ér a vezérlés, addigra az adattagoknak inicializálva kell lenniük. A törzsben ezért már inicializált értékeket írunk felül, ami erőforrás pazarlás. Primitív típusok esetén ez nem jelent problémát, összetett típusok esetén viszont számottevő lehet.

17.3.1. Megjegyzés. A konstruktor törzsében történő értékadás 2 lépés (mivel előtte egy alapértelmezett konstruktor már inicializálta az objektumot), az inicializálás csak 1.

Fontos megjegyzés, hogy az a struktúra elemei a mezők definiálásának a sorrendjében inicializálódnak. Tehát, bármilyen sorrendben írjuk mi az inicializációs listát, mindig először a `data`, és utána a `next` fog inicializálódni.

Előfordulhat olyan, hogy szeretnénk létrehozni egy listát, de azt szeretnénk, hogy élettartama mennél minimálisabb legyen. Ezt megtehetjük például úgy, hogy ha egy külön blokkban hozzuk létre.

```
void printFirstElement(const List &l) { std::cout << l.data << std::endl; }

int main
{
    //...
    {
        List l(4);
        printFirstElement(l);
    } // l megsemmisül
    //...
}
```

Ha valóban csak ideiglenesen van szükségünk erre a listára azonban, létrehozhatunk egy temporális változót is, vagy más néven egy **név nélküli temporális változót**. Ehhez csupán annyit kell tennünk, hogy elhagyjuk a változó nevét, és a típus után egyből a konstruktor paramétereit adjuk meg.

```
List(4); //amint létrejön ez a változó, meg is fog semmisülni.
//...
printFirstElement(List(4)); //a lista élettartama a függvényhívástól a függ-
    vény futásának végéig tart.
```

17.3.2. Megjegyzés. Név nélküli temporális változó az is, ha egy literált írunk. Ha pl. az `f` függvény egy darab `int`-et vár paraméterül, akkor `f(5)` hívásakor `5` egy név nélküli temporális változó.

17.3.3. Megjegyzés. A temporális változók jobb értékek, így csak konstans referenciával, vagy érték szerint tudjuk őket átvenni.

17.4. Destrukorok

Ahogy gondoskodtunk a listaelemek létrehozásáról, gondoskodhatnánk annak megfelelő megsemmisüléséről is.

```
struct List
{
    List(int data, List *next = 0) : data(data), next(next) {}
    ~List() // dtor
    {
        delete next;
    }

    int data;
    List *next;
};
```

Az fenti tagfüggvényt, melynél a hullámvonalat közvetlenül a struktúra neve követi **destruktor**nak (*destruktor*, röviden *dtor*) nevezzük. A destruktor mindig az objektum élettartamának végén hívódik meg, és gondoskodik a megfelelő erőforrások felszabadításáról.

A destruktort is rekurzívan írtuk meg: a **next** által mutatott memóriaterület felszabadításakor meghívja `List` típusú elem destruktorát. A lista végén a `next` egy nullpointer, azon a delete hívás nem csinál semmit.

Teszteljünk!

```
int main()
{
    List head(8);
    add(&head, 7);
    add(&head, 2);
}
```

Most úgy alakítottuk át a kódot, hogy amikor létrehozzuk a listát, akkor a fejelemet a stacken hozzuk létre, melynek értéke 8, és a pointer része nullpointer. Később az `add` függvénnyel létrehozunk a heapen egy olyan listaelemet, mely 7-et tárol, és pointer része nullpointer, és az eredeti lista fejét ráállítjuk erre.

Itt sikeresen elértük, hogy a lista első eleme a stack-en, de minden más eleme a heap-en legyen. Mivel olyan struktrát írtunk, mely gondoskodik arról, hogy minden dinamikusan lefoglalt területet felszabadítson, mindent csak egyszer töröl, jó sorrendben, egy *RAII* (*Resource acquisition is initialization*) osztályt írtunk. Ez Bjarnenek egy elég szerencsétlenül választott acronymje. A lényege, hogy az adott osztály a megfelelő erőforrásokat lefoglalja magának, majd a destruktor gondoskodik az erőforrások felszabadításáról. Minden erőforrást egy stack-en lévő objektumhoz kötünk, mivel azok garantáltan automatikusan fel fognak szabadulni, a destruktoruk le fog futni. Jelen esetben a lista fejeleme, ami a stack-en van, felelős azért, hogy a heap-re allokált listaelemek felszabaduljanak a program futásának a végeztével. Így a felhasználónak már a free hívásra sem kell figyelnie. Bjarne híres mondása, hogy a C++ szemétgyüjtéssel rendelkező nyelv, mert nem generál szemetet. A jól megírt objektumok mindig eltakarítanak maguk után.

A konstruktor/destruktor használata ugyanolyan hatékony, mintha kézzel kezeltük volna a memóriát.

Csináljunk az `add` függvényből tagfüggvényt!

```
struct List
{
  // ...
    void add(int data) //eltűnt egy paraméter!
    {
        if (next == 0)
        {
            next = new List(data);
        }
        else
        {
            next->add(data);
        }
    }
    int data;
    List *next;
};

int main()
{
    List head(8);
    head.add(7);
    head.add(2);
}
```

A nyelv egyik szépsége, hogy a felhasználónak nem kell tudnia, hogy hogyan reprezentáltuk a listát. A listát az a felhasználó is tudja használni, aki nem ismeri a heap-et, nem hallott még soha láncolt adatszerkezetekről. A későbbiekben a lista prerezentációja kicserélhető akár egy vektor szerű adatszerkezetre anélkül, hogy a felhasználói kódot módosítani kellene.

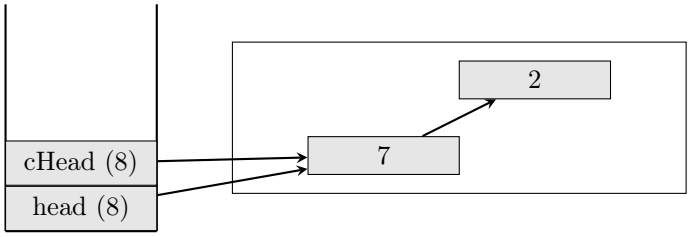
17.5. Másoló konstruktor

A fordító sok kódot generál a structunkba: konstruktoron és destruktoron kívül még **másoló konstruktor**t (*copy constructor*) is. A másoló konstruktor egy olyan konstrukor, melynek egyetlen paramétere egy azonos típusú objektum. Ez alapértelmezetten minden adattagot lemásol az adott adattag másoló konstruktora segítségével. Primitív típusoknál ez bitről bitre másolást jelent. Mi ennek a következménye?

```
int main()
{
    List head(8);
    head.add(7);
    head.add(2);
    {
        List cHead = head;
    } //itt lefut cHead destruktora
}
```

Fentebb létrehoztunk egy új listát `head` mintájára. A másolatnak a destruktora hamarabb lefut. Ha sanitizerrel fordítunk, futáskor hibaüzenetet kapunk: felszabadított memóriaterületet szeretnénk használni. Ennek az az oka, hogy a `cHead`-ben lévő pointer **ugyanarra** a listára fog mutatni (lévén a bitről bitre történő másolás történt a pointernél). A `cHead` megsemmisülése után a `head` destruktora megpróbál beleolvasni a már felszabadított memóriaterületbe.
A megoldás egy saját másoló konstruktor bevezetése!

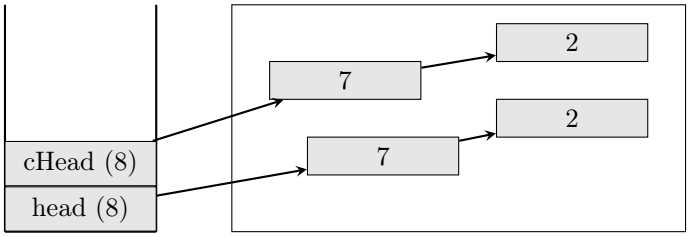
```
struct List
{
    //...

    List(const List &other) : data(other.data), next(0)
    {
```



Zárójelben a lista első elemének `data` adattagjának értéke.

```
        if (other.next != 0)
        {
            next = new List(*other.next);
        }
    }

    //...
};

int main()
{
    List head(8);
    head.add(7);
    head.add(2);
    {
        List cHead = head;
    }
}
```

Mint a korábbi függvényeink, ez is rekurzív: a `new List(*other.next)` újra meghívja a copy konstruktort, ha az other.next nem nullpointer.



Ezzel meg is oldottuk a problémát.
Figyelem, ez egy **copy konstruktor, nem értékadás operátor!** Itt a `cHead` még nincs létrehozva, amikor head-el **inicializáljuk**. Ha az egyenlőségjel bal oldalán lévő objektum még nem jött létre, mint itt, akor a copy konstruktor hívás történik. Ellenkező esetben értékadás operátor.

```
List cHead = head; //copy ctor

List cHead;
cHead = head; //értekadás
```

17.6. Értékadó operátor

Az előző órán elkészült másoló konstruktor megoldotta a probléma egy részét, de az értékadással hasonló problémák merülnek fel. Hasonlóan a fordító által generált másoló konstruktorhoz, az alapértelmezetten az értékadás operátor (*assignment operator*, vagy röviden =) is meghívja az egyes tagok értékadó operátorait, amik a primitív típusok esetén bitről bitre másolnak.

```
struct List
{
```

```
    // ...

    // copy constructor
    List(const List &other) : data(other.data), next(0)
    {
        if (other.next != 0)
            next = new List(*other.next);
    }

    // assignment operator
    List& operator=(const List &other)
    {
        delete next;
        data = other.data;
        if (other.next) // if(other.next) == if(other.next != 0)
            next = new List(*other.next);
        else
            next = 0;
        return *this;
    }

    //...
};
```

Az új értékadás operátor először kitörli az aktuális lista elemeit, majd rekurzív módon lemásolja az `other` elemeit.
Kicsit részletesebben: az első lépés a `delete next;`, a fejelemet leszámítva felszabadítja az összes többi listaelemhez tartozó memóriaterületet, eztán lemásoljuk az `other` fejelemében lévő adatot. Végül a másoló konstruktorhoz hasonlóan rekurzív módon lemásoljuk az `other` farokrészét (amennyiben az létezik).
Azáltal, hogy egy listaelem másolásakor az összes listaelem által birtokolt (heapen lévő) objektum is másolásra kerül, elértük azt, hogy a másolaton végzett módosításoknak ne legyen hatása az eredeti objektumra.

Figyeljük meg, hogy az értékadás operátorban egy az adott objektumra mutató referenciával tértünk vissza. Ennek oka az, hogy szeretnénk, ha lehetséges lenne az értékadások láncolása:

```
            int a = b = c = d = 0; // a = (b = (c = (d = 0)))
```

Itt rendre (az operátorok kiértékelési sorrendjét tekintve) `d`-re, `c`-re, `b`-re, `a`-ra mutató referenciát ad vissza az értékadás operátor, így a végeredmény az lesz, hogy mindegyik változót 0-ra inicializáltuk.

17.6.1. Megjegyzés. Miért nem konstans referenciával térünk vissza? Akkor nem csinálhatnánk hasonlót: (legyen `f(int &)`) `f(a = 0)`.

Egyszerűsítsük a lista kiírását is!

```
struct List
{
    //...

    void print ()
    {
        std::cout << data << ' ';
        if (next)
            next->print();
    }

    //...
};
```

Ellenőrizzük, hogy az eredeti elemek nem változtak a másolás következtében!

```
int main ()
{
```

```
    List head(7);
    head.add(8);
    head.add(2);
    {
        List cHead = head;
    }
    head.print();
}
```

Kimenet: `7 8 2`
Elértük, hogy biztonságos legyen a lista használata? Most már látszólag a másolás és az értékadás sem okozhat memóriakezeléssel kapcsolatos problémát. Van egy eset, amire a kód még nincs felkészítve! Mi történik, ha önmagának adjuk értékül a listát? Az első lépésben törlésre fog kerülni a fejelemet leszámítva minden elem. Eztán a következő lépésekben már felszabadított memóriaterületetről próbáljuk átmásolni a megfelelő adatokat. A `cHead = cHead` tehát use-after-free hibát fog okozni. Mennyire valós a félelem ettől a hibától? Hiszen nem gyakran adunk értékül egy objektumot önmagának! Tekintsük a következő függvényt:

```
void f(List &l1, List &l2)
{
    //...
    l1 = l2;
    //...
}
```

Ha valaki ugyanazt a listát adja meg mindkét paraméternek, akkor máris megvan a baj. Látható tehát, hogy ilyen jellegű hibát nem feltétlen olyan nehéz elkövetni. Módosítsuk az értékadás operátorunkat oly módon, hogy ne okozzon problémát az önértékadás:

```
List& operator=(const List &other)
{
    if (this == &other) return *this;
    delete next;
    if (other.next)
        next = new List(*other.next);
    else
        next = 0;
    return *this;
}
```

Emlékezzünk, a `this` kulcsszó segítségével tudunk rámutatni tagfüggvényen belül az adott objektumra, amin a tagfüggvény meghívásra került. Tulajdonképpen arról van szó, hogy például a `head.print()` esetében enélkül a kulcsszó nélkül nem tudnánk `print()`-en belül `head`-re hivatkozni. Az objektum neve tagfüggvényen belül nem elérhető. A `this` egy pointerként is felfogható, mely a `head` objektumra mutat.

Sikeresen létrehoztunk egy félig *reguláris típust*.

Egy `T` típus **reguláris típus**, ha van neki:

1. default konstruktora
2. destruktora
3. értékadás operátora, továbbá

   ```
   T b;
   T a;
   a = b;
   ```

   Ekkor `a` ekvivalens `b`-vel, és ha `a`-t módosítjuk, akkor `b` nem változik, és viszont.
4. másoló konstruktora, továbbá

   ```
   T b;
   T a = b;
   ```

   Ekkor `a` ekvivalens `b`-vel, és ha `a`-t módosítjuk, akkor `b` nem változik, és viszont.
5. egyenlőségvizsgálat (`operator==`) (Ha nincs, akkor az adott típust félig regulárisnak nevezzük)

17.7. Adattagok védettsége

Biztonságossá sikerült tenni a lista használatát? A felhasználó most már a megfelelő metódusok, valamint a másolás és értékadás használatával nem tud elkövetni memóriakezeléssel kapcsolatos hibát. Az adattagokhoz azonban továbbra is hozzá fér. Így, a következő módon továbbra is érhetik meglepetések:

```
int main()
{
    List head(8);
    head.add(7);
    head.add(2);
    head.next = 0;
}
```

Itt az első listaelem utáni elemeket lecsatoljuk, és azoknak a memóriája elszivárog. A probléma forrása az, hogy a **felhasználó hozzáfér az adattagokhoz.** A felhasználó így olyan állapotba állíthatja az adott objektumot, amire az objektum tagfüggvényei nincsenek felkészülve. Másik probléma, hogy a felhasználó azáltal, hogy hozzáfér az adattagokhoz, ki tudja használni a lista belső reprezentációját. Emiatt nehezebben fenntarthatóvá válik a kód, mivel a lista reprezentációját érintő változtatások esetén a felhasználói kódokat is módosítani kell. Példaképp, ha a `List` `next` adattagjának új nevet adunk, akkor át kéne írni minden olyan kódot ami `next`re hivatkozik.
Jó lenne, ha hiába változna a belső reprezentáció, a kódnak nem kéne a felhasználói kódnak változnia.
Rejtsük el az adattagokat a felhasználó elől!

```
class List // lehetne struct is
{
public:
    List(int data, List *next = 0) : //...
    void add(int data) {//...}
    List(const List &other) //...
    List& operator=(const List &other) {//...}
private:
    int data;
    List *next;
};
```

Az osztály minden tagja, mely a `public` kulcsszó után jön, elérhető bárki számára. A `private` kulcsszó után következő adatok csak is kizárólag az adott osztályon belül érhető el (leszámítva a `friend`eket, de erről később). A `class` és a `struct` abban tér el, hogy alapértelmezetten a `struct` minden tagja publikus, míg `class`-nak privát. A `public` ill. `private` kulcsszóval mind a kettőnél explicit megadhatjuk a láthatóságot.

Van egy súlyos következménye az imént bevezetett adatrejtésnek. Például, így nem tudunk hozzáférni a második elemhez kívülről. Vagy bármelyik másikhoz. Ilyenkor a naív megoldás az, ha módosítjuk a listát, és minden olyan függvényt, aminek hozzá kéne férnie az elemekhez, metódussá tesszük. Hiszen a lista metódusai hozzáférnek a privát adattagokhoz is.
Ezt ugyanakkor nem akarjuk minden egyes függvénnyel megtenni. A végén nagyra nőne a lista osztály, ami nagyban rontja az kód érthetőségét és fenntarthatóságát. Emellett a reprezentáció változtatásával megint sok függvényt kellene módosítani.
Írjunk felsorolót, amivel hozzáférhetünk az elemekhez kívülről is.

```
class List
{
    //...

    void First()
    {
        cursor = this;
    }
    int& Current()
    {
        return cursor->data;
    }
```

```
    void Next()
    {
        cursor = cursor->next;
    }
    bool End()
    {
        return cursor == 0;
    }

private:
    int data;
    List *next;
    List *cursor;
};
```

Figyeljük meg, hogy `Current()` referenciával tér vissza, hogy ne `data` másolatát, hanem a `data`-ra hivatkozó referenciát kapjunk meg, így tudjuk a lista elemeinek az értékeit módosítani.

Első látásra a problémát orvosoltuk, tudunk a listákkal tárolt adatokkal dolgozni (pl. összeadni őket, stb). De egy rendezésnél már problémásabb. Ha csak egy kurzorunk van, nem tudunk egyszerre 2 elemhez hozzáférni, hogy összehasonlítsuk őket vagy megcseréljük őket. Egy lehetséges megoldás:

```
class List
{
    //...

    void First()
    {
        cursor = this;
    }
    int& Current()
    {
        return cursor->data;
    }
    void Next()
    {
        cursor = cursor->next;
    }
    bool End()
    {
        return cursor == 0;
    }

    void First2()
    {
        cursor2 = this;
    }
    int& Current2()
    {
        return cursor2->data;
    }
    void Next2()
    {
        cursor2 = cursor2->next;
    }
    bool End2()
    {
        return cursor2 == 0;
    }
```

```
private:
    int data;
    List *next;
    List *cursor;
    List *cursor2;
};
```

Ezzel már egy rendezést meg tudunk oldani. De talán érezhető, hogy ez nem a legszebb megoldás. Ha három `cursor` kéne egy algoritmushoz, akkor megint bajban lennénk. Ennél létezik sokkal szofisztikáltabb megoldás.

17.8. Iterátorok

Az iterátorok a pointerek általánosításai, segítségükel tudunk végigiterálni egy konténer elemein (azaz velük tudjuk lekérdezni a konténer elemeit). Ehhez szükségünk van arra, hogy tudjuk hol kezdődik és végződik a konténerünk. E láncolt lista esetében hozzá tudunk férni az első elemhez (az lesz ugye `head` a fenti példákban) és tudjuk hogy mindig nullpointerrel az utolsó elemben lévő mutató értéke. Legyen az iterátunk neve `Iterator`!

```
class List
{
    //...
public:
    Iterator begin()
    {
        return Iterator(this);
    }
    Iterator end()
    {
        return Iterator(0);
    }
};
```

E két metódus segítségével már meg tudjuk adni a lista elejét és végét! A `head.begin()` vissza fog adni egy iterátort, ami a `head` a legelső elemére mutat, `head.end()` az utolsó utáni elem. Már csak magát az iterátort kell megírni, mely egy un. *forward iterator* lesz.

Egy `T` típus **forward iterator**, ha rendelkezik:

1. `++` operátorral
2. egyenlőség vizsgáló operátorral `==`
3. egyenlőtlenség vizsgáló operátorral `!=`
4. dereferáló operátorral `*`

De hova írjuk az `Iterator` osztályt? Hiszen ha a lista elé tesszük, az `Iterator` nem fogja tudni, hogy mi az a `List`. Ha utána tesszük, nem fogja tudni a `List` hogy mi az az `Iterator`. Erre szükség van a Listában az `end` és a `begin` metódus megírásához. A körkörös függőség feloldása érdekében **forward deklarálni** fogunk.

```
class List; //forward declaration

class Iterator
{
public:
    Iterator(List *_p) : p(_p) {}
    bool operator==(Iterator &other)
    {
        return p == other.p;
    }
    bool operator!=(Iterator &other)
    {
        return !(*this == other);
    }
```

```
    //...
private:
    List *p; //(*)
};

class List {//...};
```

A fordító ezek után a csillaggal jelölt sorban nem fog arra panaszkodni, hogy a `List` egy ismeretlen azonosító. A forward deklaráció azt mondja a fordítónak, hogy a `List` később, akár egy másik fordítási egységben definiálva lesz. Amennyiben nem akarjuk dereferálni `p`-t, elég csupán a forward deklarálni!

Még hiányzik a `++` és a `*` operátor.

```
class List;

class Iterator
{
public:
    Iterator(List *_p) : p(_p) {}
    bool operator==(Iterator &other)
    {
        return p == other.p;
    }
    bool operator!=(Iterator &other)
    {
        return !(*this == other);
    }
    Iterator operator++()
    {
        p = p->next;
        return *this;
    }
    int& operator*()
    {
        return p->data;
    }
private:
    List *p;
};

class List {//...};
```

Amennyiben az előbb említett operátorokat az `Iterátor`on belül fejtjük ki, a fordító dob egy hibaüzenetet, miszerint a `List` egy un. *incomplete type*. A forward deklaráció miatt a fordító tudja már, hogy a `List` az egy osztály. De nem tudja ellenőrizni, hogy `next` adattaggal rendelkezik-e. Ezért ezeket a tagfüggvényeket a `List` kifejtése után kell írni.

```
class List;

class Iterator
{
public:
    Iterator(List *_p) : p(_p) {}
    bool operator==(Iterator &other) const
    {
        return p == other.p;
    }
    bool operator!=(Iterator &other) const
    {
        return !(*this == other);
    }
```

```
    Iterator operator++();
    int& operator*();
private:
    List *p;
};

class List {//...};

Iterator Iterator::operator++()
{
    p = p->next;
    return *this;
}
int& Iterator::operator*()
{
    return p->data;
}
```

Felmerül még egy probléma: a `next` és `data` private adattagok. Az `Iterator` nem fér hozzá! Erre megoldás, ha **barát** (*friend*) osztállya lesz az Iterátor a `List`nek.

```
class List
{
    //...
    friend class Iterator;
    //...
}
```

A barátként deklarált osztályok és függvények hozzá tudnak férni az osztály privát adattagjaihoz is.

Az így kapott iterátorunkkal be tudjuk járni a lista elemeit, hozzá tudunk férni a listában tárolt adatokhoz. Így már a `print()`-et tagfüggvény helyett szabad függvényként is megírhatjuk. Amennyiben lehetőségünk van egy függvényt szabad (osztályon kívüli) függvényként megírni, érdemes élni a lehetőséggel. Ezáltal az osztályaink kisebbek és könnyebben érthetőek lesznek.

```
void print(List &l)
{
    for(Iterator it = l.begin(); it != l.end(); ++it)
    {
        std::cout << *it << ' ';
    }
    std::cout << std::endl;
}
```

Ez a függvény, csak úgy mint az *STL* függvények is, balról zárt, jobbról nyitott [ ) intervallummal dolgozik. Azaz az `end()` már nem eleme a listának, az az utolsó utáni elem (*past-the-end iterator*).

Nézzük meg, hogy miért jobb az iterátor, mint a felsoroló. Annyi `Iterator`-t hozunk létre, amennyit csak akarunk, nem vagyunk korlátozva a `cursor`ok darabszáma által. Az iterátorok emellett tetszőlegesen tárolhatóak, átadhatóak függvényeknek. Nincs elrejtve az objektum belsejébe. Továbbá, tudjuk módosítani a listában található elemeket anélkül, hogy a lista reprezentációját ismernünk kellene. Egy iterátorokat használó kód módosítása nélkül megváltoztatható a lista belső reprezentációja. Ez lehetővé teszi azt, hogy több programozó csapatban dolgozzon, egymástól független kódrészleteket úgy tudjon módosítani, hogy ne kelljen egymás kódját átírni.

Megfigyelhető, hogy `print` nem módosítja a paraméterként kapott listát, így átvehetnénk konstansk referenciával is. Ennek az eredménye fordítási hiba: az `end` és `begin` nem konstans metódusok, hiszen ami iterátort visszaad, azokon keresztül tudjuk módosítani az objektumot.

17.9. Konstans iterátorok

Erre a megoldás, ha konstans iterátort is írunk, ami egy konstansra mutató mutató általánosítása.

```
class List;
```

```
class Iterator
{
    //...
};

class ConstIterator
{
public:
    Const Iterator(const List *_p) : p(_p) {}
    bool operator==(ConstIterator &other) const
    {
        return p == other.p;
    }
    bool operator!=(ConstIterator &other) const
    {
        return !(*this == other);
    }
    ConstIterator operator++();
    int operator*() const; //nem referenciával tér vissza!
private:
    const List *p; //konstanra mutató pointer!
};

class List
{
    //...
    ConstIterator begin() const
    {
        return ConstIterator(this);
    }
    ConstIterator end() const
    {
        return ConstIterator(0);
    }

    friend class ConstIterator;
    //...
};

Iterator Iterator::operator++() {//...}
int& Iterator::operator*() {//...}

ConstIterator ConstIterator::operator++()
{
    p = p->next;
    return *this;
}
int ConstIterator::operator*() const
{
    return p->data;
}
```

Azon metódusok, amik után `const` kulcsszó szerepel (ld. fenti példa) nem tudják megváltoztatni az adott osztály adattagjait. Ezeket a metódusokat **konstans metódusok**nak (*const method*) hívják. Egy konstans objektumon csak akkor lehet meghívni egy metódust, ha az a metódus konstans.

17.9.1. Megjegyzés. Ami lehet `const`, az **legyen** `const`!

Egy `const List` típusú objektumon csak a konstans metódusok hívhatóak meg, így a meghívott `begin()`

visszatérési értékének típusa `ConstIterator` lesz. Egy nem konstans `List` objektumnál viszont `Iterator` lesz.

```
int main()
{
    List head(8);
    head.add(7);
    head.add(2);
    {
        ConstIterator cit = head.begin(); //hiba
    }
}
```

Két különböző típust nem tudunk egymásnak értékül adni, ha nem létezik köztük konverzió. Mivel nem konstansra mutató mutató konvertálódhat konstansra mutató mutatóvá, ezért természetes lenne hasonló konverziót az iterátorok közt is bevezetni. Ezt egy új konstruktor segítségével tehetjük meg:

```
class ConstIterator
{
    //...
public:
    ConstIterator(Iterator &other) : p(other.p) {}
    //...
}
```

Emlékezzünk, hogy az `Iterator`-nak a `p` adattagja privát. A forduló kódhoz egy friend deklaráció bevezetése szükséges!
Meglepő lehet, hogy az értékadás egy új konstruktor bevezetését követően le fog fordulni. A `head.begin()` a fenti esetben egy `Iterator`-t ad vissza, mivel `head` nem konstans! Eztán a fordító az imént megírt konstruktor segítségével az `Iterator` típusú változóból készít egy `ConstIterator` típusút. Ezzel *implicit kódon átkonvertálja* `Iterator`-t `ConstIterator`-rá, és utána hívja meg a másoló konstruktort.

17.10. Konverziós operátor

Más módon is át lehet konvertálni egy `Iterator` típusú objektumot `ConstIterator` típusúvá. Azzal, hogy létrehozunk egy un. **konverziós operátor**t (*conversion operator* vagy *user defined conversion*).

```
// ...
class ConstIterator; //forward deklaráció itt szükséges

class Iterator
{
public:
    // ...
    operator ConstIterator() const; //konverziós operátor
    // ...
};

class ConstIterator{//...}

Iterator::operator ConstIterator() const
{
    return ConstIterator(p);
}
// ...
```

A fenti konverziós operátor segítségével implicit módon végre lehet hajtani a konverziót.

```
ConstIterator cit2 = head.begin(); //ok
```

Amennyiben azt szeretnénk, hogy a konverzió létezzen, de implicit módon ne jöjjön létre, használhatjuk az `explicit` kulcsszót. Ez a kulcsszó azonban csak C++11 óta használható ilyen módon:

```
class Iterator
```

```
{
public:
    // ...
    explicit operator ConstIterator() const;
    // ...
};
```

Nézzünk meg pár példát, hogyan lehet a konverziót explicit módon használni:

```
int main()
{
    List head(8);
    ConstIterator cit1 = head.begin(); // hiba
    ConstIterator cit2 = ConstIterator(head.begin()); // ok
    ConstIterator cit3 = static_cast<ConstIterator>(head.begin()); // ok
}
```

17.10.1. Megjegyzés. A fent látható `static_cast`-ról később lesz részletesen szó.

17.10.2. Megjegyzés. A fenti módosítások ismét csak gyakorlás célját képezték, az elkészítedő listának nem lesz része a konverziós operátor.

17.10.3. Megjegyzés. Amikor lehetőségünk van rá, használjunk egy paraméteres konstruktorokat a konverzióra. Amennyiben ahhoz az osztályhoz, amivé konvertálni szeretnénk, nem adhatunk hozzá új konstruktort (például könyvtári osztály), akkor használjuk a konverziós operátort.

17.11. Explicit konstruktorok

Azt, hogy az egy paraméteres konstruktorokat a fordító implicit konverzióra felhasználja a fenti módon, megtilthatjuk itt is, ha `explicit` kulcsszót írunk a konstruktor elé.

```
class ConstIterator
{
    //...
public:
    explicit ConstIterator(Iterator &other) : p(other.p) {}
    //...
}
```

Jelen esetben nem akarjuk a konverziót megtiltani, lévén az `Iterator` és a `ConstIterator` hasonló feladatot lát el. Nézzük meg az eddigi kódjainket. Okozhat valahol meglepetést egy konverzió? A `List` egyik konstruktra gyanús lehet. Ha egy függvény listát vár paraméterül, egy darab `int`-et is elfogad, hisz a `List` konstruktorát felhasználva tudott volna csinálni abból az `int`-ből egy egy elemű listát! Ezt elkerülendő, tegyük a `List` konstruktorát `explicit`-té.

```
class List
{
    //...
    explicit List(const int _data, List *_next = 0) : data(_data), next(
        _next) {}
    //...
}
```

18. Template

18.1. Függvény template-ek

Térjünk vissza a régebben megírt swap függvényünkhöz.

```
void swap(int &a, int &b)
{
    int tmp = a;
```

```
    a = b;
    b = tmp;
}
```

Ahogy azt láttuk, túl tudjuk terhelni ezt a függvényt, hogy más típusú objektumokat is meg tudjunk cserélni. Azonban gyorsan megállapítható, hogy állandóan egy újabb overloadot létrehozni nem épp ideális megoldás. Ez a kisebb gond, a nagyobb az, hogy a kódismétlés áldozatai leszünk: ha bármi miatt megváltozna a `swap` belső implementációja (pl. találunk hatékonyabb megoldást), az összes létező swap függvényben meg kéne ejteni a változtatást. E probléma elkerülésére egy megoldás lehet, ha létrehozunk egy sablont, melynek mintájára a fordító maga tud generálni egy megfelelő függvényt.

```
template <typename T>
void swap(T &a, T &b)
{
    T tmp = a;
    a = b;
    b = tmp;
}
```

Az így implementált `swap` függvény egy *template*, és a template paramétere (`T`) egy típus. Ez alapján a fordító már létre tud hozni hozni megfelelő függvényeket:

```
int main()
{
    int a = 2, b = 3;
    swap<int>(a, b);

    double c = 1.3, d = 7.8;
    swap<double>(c, d);
}
```

A fordítónak csak annyi dolga van, hogy minden `T`-t lecseréljen `int`-re, és már kész is a függvény. A fenti példában mi explicit megmondtuk a fordítónak, hogy `swap`-ot milyen template paraméterrel példányosítsa (*instantiate*), azonban függvényeknél erre nem feltétlenül van szükség: a fordító tudja `a` és `b` típusát, így ki tudja találni hogy mit kell behelyettesítenie.

```
int main()
{
    int a = 2, b = 3;
    swap(a, b);

    double c = 1.3, d = 7.8;
    swap(c, d);
}
```

Ezt a folyamatot (amikor a fordító kitalálja a tempalte paramétert) **template paraméter dedukciónak** (*template parameter deduction*) hívjuk.

18.1.1. Megjegyzés. Természetesen a példányosítás jóval bonyolultabb annál, hogy a a fordító minden `T`-t egy konkrét típusra cserél, de ebben a könyvben az egyszerűség kedvéért elégedjünk meg ennyivel.

Nem csak típus lehet template paraméter – bármi ami **nem** karakterlánc literál vagy lebegőpontos szám.

18.1.2. Megjegyzés. A lebegőpontos számokra vonatkozó indoklást később a template specializációknál lesz leírva, de a karakterlánc literálokra már most adhatunk választ.
Mivel a C++ban van lehetőség függvénytúlterhelésre, ezért a fordító fordítás közben nem csak a függvény nevét, de annak paraméterlistáját, visszatérési értékét és a template paraméterekre vonatkozó információkat (stb.) is kénytelen eltárolni a függvény nevével együtt. Amennyiben karakterlánc literál is lehetne template paraméter, nagyon meg tudna nőni ennek a sztringnek a hossza, és lassíthatná a fordítási időt.

```
template <typename T, int ArraySize>
int arraySize(const T (&array)[ArraySize])
{
```

```
    return ArraySize;
}

int main()
{
    int i[10];
    std::cout << arraySize(i) << std::endl; //10
}
```

A fenti kód a 3. gyakorlat végén tett megjegyzésből lehet ismerős. Jól demonstrálja a template paraméter dedukciót.

18.2. Osztály template-ek

Nem csak függvények, osztályok is lehetnek template-ek melyen nagyon hasonlóan működnek. A következő kódrészletekben a template osztályok mellett megismerkedhetünk még a template-ek „lustaságával" is.

```
#include <iostream>

template <typename T>
struct X
{
    void f()
    {
        T t;
        t.foo();
    }
};
struct Y
{
    void bar() {}
};
int main() {}
```

Ez a kód úgy tűnhet, hogy nem fog lefordulni, lévén mi soha semmilyen `foo` tagfüggvényt nem írtunk, azonban mégis le fog. Ez azért van, mert a template osztályok (és függvények) csak sablonok, amiből aminek alapján a fordító generálhat egy konkrét osztályt (vagy függvényt), és mivel sose példányosítottuk, fordítás után az `X` template osztály nem fog szerepelni a kódban. Szintaktikus ellenőrzést végez a fordító, (pl. zárójelek be vannak-e zárva, pontosvessző nem hiányzik-e stb.), de azt, hogy van-e olyan `T` típus, ami rendelkezik `foo()` függvénnyel, már nem.

Példányosítsuk az `X` osztályt `Y`-nal!

```
int main()
{
    X<Y> x;
}
```

Ekkor már azt várnánk hogy fordítási hibát dobjon a fordító, hisz `Y`-nak nincs `foo()` metódusa, azonban mégis gond nélkül lefordul, mivel az `f()` tagfüggvényt nem hívtuk meg, így nem is példányosult az osztályon belül.

```
int main()
{
    X<Y> x;
    x.f();
}
```

Itt már végre kapunk fordítási hibát, mert példányosul `f()`. Ez jól mutatja, hogy a template-ek lusták, és csak akkor példányosulnak, ha „nagyon muszáj".

A template-eknek adhatunk meg alapértelmezett értéket.

```
template <typename T = void> //alapértelmezett paraméter
struct X {/* ... */};
```

```
struct Y {/* ... */};

int main()
{
    X<Y> x;
    X<> x2;
}
```

Ilyenkor nem szükséges megadni template paramétert (mely esetben értelemszerűen `X<> == X<void>`).
Ahogyan az említve volt korábban, szinte bármi lehet template paraméter, akár egy másik template is.

```
template <typename T>
struct X {/* ... */};

struct Y {/* ... */};

template <template <typename> class Templ>
struct Z
{
    Templ<int> t;
};

int main()
{
    Z<X> z;
}
```

Fent `Templ` egy olyan template, aminek a template paramétere egy típus. Így `Z`-nek a template paramétere egy olyan template, aminek a template paramétere egy típus. Mivel `X` egy template (és template paramétere egy típus), így megadható `Z`-nek template paraméterként.

18.2.1. Megjegyzés. Fent a template paraméter listában `typename` helyett `class` szerepel. Ezek gyakorlatilag ekvivalensek, mind a kettő azt jelenti, hogy az adott paraméter típus (bár a `typename` beszédesebb).

A fenti példákban mindig egy default konstruktort hívtunk meg, amikor objektumokat hoztunk létre. Helyes lenne-e az, ha explicit módon kíírnánk a zárójeleket (hangsúlyozva a default konstruktor hívást)?

```
int main()
{
    X<Y> x();
    X<> y2();
    Z<X> z();
}
```

A kód helyesen lefordul, de a jelentése nem ugyanaz, mintha nem lenne ott a zárójel. Mivel a c++ nyelvtana nem egyértelmű, más kontextusban ugyanaz a kódrészlet mást jelenthet (egyik legegyszerűbb példa a `static` kulcsszó), így meg kellett alkotni egy olyan szabályt, miszerint amit deklarációként lehet értelmezni, azt deklarációként **kell** értelmezni. Igy ezek függvénydeklarációk lesznek: Az első esetben például egy olyan függvényt deklarálunk, melynek neve `x`, `X<Y>`-al tér vissza és nem vár paramétert.
Így ha default konstruktort szeretnék meghívni, semmilyen zárójelt nem szabad használni.

18.2.2. Megjegyzés. C++11ben lehet gömbölyű zárójel helyett helyett kapcsos zárójelet alkalmazni konstruktorhívásnál, így ez a probléma nem fordulhat elő. pl: `X<Y> x{};`

A template-ek paramérének ismertnek kell lennie fordítási időben.

```
template <int N>
void f() {}

int main()
{
    int n;
```

```
    std::cin >> n;
    f<n>(); //hiba, n nem ismert fordítási időben
}
```

Ez nyilvánvaló, hisz a template-eknek az a funkciója, hogy a fordító generáljon belőlük példányokat, és a fordítási idő végeztével erre nincs lehetőség.

18.2.3. Megjegyzés. Fontos még, hogy a template-ek nagyon megnövelik a fordítási időt, így nem mindig éri meg egy olyan függvényt is template-ként megírni, melyet nem feltétlenül muszáj.

18.3. Template specializáció

Néha szeretnénk, hogy bizonyos speciális behelyettesítéseknél más legyen az implementáció mint az alap sablonban. Ilyenkor szokás **specializációkat** (*template specialization*) létrehozni:

```
template <class T>
struct A
{
    A() { std::cout << "general A" << std::endl; }
};

template <> //template specializáció
struct A<int>
{
    A() { std::cout << "special A" << std::endl; }
};

template <class T>
void f() { std::cout << "general f" << std::endl; }

template<> //template specializáció
void f<int>() { std::cout << "special f" << std::endl; }

int main()
{
    A<std::string> a1; //general A
    f<std::string>(); //general f
    A<int> a2; //special A
    f<int>(); //special f
}
```

Mind `A` osztályhoz, mind `f` függvényhez létrehoztunk egy specializációt arra az esetre, ha a template paraméterként `int`-et kapnak. Számos okunk lehet arra hogy ezt tegyük: a standard könyvtár megfényesebb példája az `std::vector` osztály, mely egy template, és van template specializációja `bool` esetre.

18.3.1. Megjegyzés. Az `std::vector<bool>` számos optimalizációkat tartalmazhat (persze nem feltétlenül, hisz ez implementáció függő): általában nem `bool`-okban tárolja az adatokat, hanem bitekben. Sajnos azonban ez hátrányokkal is jár, például hogy a `[]` operátor érték és nem referencia szerint ad vissza.

18.3.2. Megjegyzés. Visszatérve egy korábbi állításhoz, miért nem lehet lebegőpontos szám template paraméter? Lévén két lebegőpontos szám könnyedén lehet csak nagyon kis mértékben eltérő, ezért könnyű egy ilyesmi hibába belefutni:
Legyen adott `d1, d2`, fordítási időben ismert lebegőpontos szám! Mi azt hisszük, hogy ez a kettő egyenlő, de mivel számos módosításon mentek keresztül, minimális mértékben, de nem lesznek egyenlőek. Ilyenkor ha egy template paraméterként lebegőpontos számot váró függvénynek megadnánk őket template paraméterül, kétszer kéne példányosítani az adott függvényt.
Ez a kisebb gond, de mi van ha pont erre az értékre mi létrehoztunk egy template specializációt, és csak (pl.) `d1` esetében került az a függvény meghívásra? Ez ellen a fordító se tudná a felhasználót megvédeni.

Írjunk faktoriális számoló algoritmus template-ek segítségével!

```
template<int N>
struct Fact
{
    static const int val = N*Fact<N-1>::val;
};

template<>
struct Fact<0>
{
    static const int val = 1;
};

int main()
{
    std::cout << Fact<5>::val << std::endl; //120
}
```

`Fact` 4szer példányosul: `Fact<5>, ..., Fact<1>`, majd a legvégén az általunk specializált `Fact<0>`-t hívja meg.

18.3.3. Megjegyzés. Ahogyan ezt korábban megállapítottunk, egy **konstans** osztályszintű változót függvénytörzsön belül is inicializálhatunk.

Ez fel is hívja a figyelmet a template-ek veszélyeit statikus változók használatakor.

```
template <class T>
class A
{
    static int count;
public:
    A()
    {
        std::cout << ++count << ' ';
    }
};

template <class T>
int A<T>::count = 0;

int main()
{
    for(int i = 0; i<5; i++)
    {
        A<int> a;
        A<double> b;
    }
}
```

Kimenet: `1 1 2 2 3 3 4 4 5 5`
Bár arra számítanánk, hogy 1-től 10ig lesznek a számok kiírva, ne felejtsük, hogy itt két teljesen különböző osztály fog létrejönni: `A<int>` és `A<double>`, így a `count` adattag hiába osztályszintű, 2 teljesen különböző példánya lesz ennek is: `A<int>::count` és `A<double>::count`.

18.4. Dependent scope

Lehetőségünk van arra hogy osztályon belül deklaráljunk még egy osztályt. Bár erről bővebben a következő órai jegyzetben lesz szó, egy igen fontos problémát vet fel.

```
class A
{
public:
```

```
    class X {};
};

void f(A a)
{
    A::X x;
}

int main()
{
    A a;
    f(a);
}
```

Ezzel semmi probléma nincs. Legyen A egy template osztály!

```
template <class T>
class A
{
public:
    class X {};
};

template <class T>
void f(A<T> a)
{
    A<T>::X x;
}

int main()
{
    A<int> a;
    f(a);
}
```

Itt máris bajba jutottunk, a fordító azt a hibát fogja jelezni, hogy `X` egy un. **dependent scope**-ban van. Ez azt jelenti, hogy attól függően, milyen template paraméterrel példányosítjuk `A`-t, `X`-nek lehet más a jelentése. Az alábbi kód ezt jól demonstrálja:

```
template <typename T>
struct A
{
    class X{};
};

template <>
struct A <int>
{
    static int X;
};

int A<int>::X = 0;

template <typename T>
void f()
{
    A<T>::X;
}
```

Itt az `f` függvényben vajon mi lesz `A<T>::X`? A válasz az hogy nem tudni, hisz ha `int`-el példányosítunk akkor statikus adattag, ha bármi mással, akkor meg egy típus. Ezért kell a fordítónak biztosítani, hogy a template paramétertől függetlenül garantáltan típust fog oda kerülni. Ezt a `typename` kulcsszóval tehetjük meg.

```
template <typename T>
void f()
{
    typename A<T>::X;
}
```

A `typename` garantálja a fordítónak, hogy bármi is lesz `T`, `A<T>::X` mindenképpen típus lesz. Ha mégis olyan template paramétert adunk meg, aminél ez nem teljesülne (ez esetben `T = int`) akkor fordítási idejű hibát kapunk.

18.4.1. Megjegyzés. A fordító általában szokott szólni, hogy a `typename` kulcsszó hiányzik.

18.4.2. Megjegyzés. A dependent scope problémája nem csupán az osztályon belüli osztályokra érvényes. Nemsokára meglátjuk, hogy a `typedef` kulcsszó is ide tud vezetni.

Így viszont felmerülhet a kérdés hogy van-e szükség `typename` kulcsszóra, ha egy `std::vector<int>::iterator` típusú objektumot akarunk létrehozni (`std::vector<int>::iterator vit;`). A válasz az hogy nem, hisz ha konkrétan megadjuk a típust, amellyel példányosítanánk, akkor a fordító arra a konkrét típusra vissza tudja keresni, hogy `std::vector<int>::iterator` típus-e, vagy sem.

19. Header fájlra és fordításra egységre szétbontás

Visszatérve a korábban írt láncolt listánkhoz, bátran állíthatjuk, hogy mindennel rendelkezik ami számunkra fontos. Azonban ha egy darab header fájlban tárolnánk mindent, számos problémába ütköznénk. Ha több fordítási egységbe illesztenénk be a headert, fordítási idejű hibát kapnánk, hogy számos függvényt többször próbáltunk definiálni (sértenénk az ODR-t). Erre megoldás lehet, hogy a definíciókat és deklarációkat elválasztjuk: az osztályban lévő függvények deklarációit hagyjuk meg a header fájlban, és a definíciókat egy külön fordítási egységbe tegyük!

19.0.1. Megjegyzés. Feltűnhet majd, hogy pár függvénydefiníció bent maradt. Erre később lesz magyarázat.

list.hpp:

```
#ifndef LIST_H
#define LIST_H

#include <iosfwd>

class List;

class Iterator
{
public:
    explicit Iterator(List *p) : p(p) {}
    bool operator==(Iterator other) const { return p == other.p; }
    bool operator!=(Iterator other) const { return !(*this == other); }
    Iterator operator++();
    int& operator*() const;
private:
    friend class ConstIterator;
    List *p;
};

class ConstIterator
{
public:
    ConstIterator(Iterator it) : p(it.p) {}
    explicit ConstIterator(const List *p) : p(p) {}
    bool operator==(ConstIterator other) const { return p == other.p; }
    bool operator!=(ConstIterator other) const { return !(*this == other); }
```

```
    ConstIterator operator++();
    int operator*() const;
private:
    const List *p;
};

class List
{
public:
    explicit List(int data_, List *next = 0) : data(data_), next(next) {}
    ~List() { delete next; }
    List(const List &other);
    List& operator=(const List &other);
    void add(int data);
    Iterator begin() { return Iterator(this); }
    ConstIterator begin() const { return ConstIterator(this); }
    Iterator end() { return Iterator(0); }
    ConstIterator end() const { return ConstIterator(0); }
private:
    friend Iterator;
    friend ConstIterator;
    int data;
    List *next;
};

#endif
```

list.cpp:

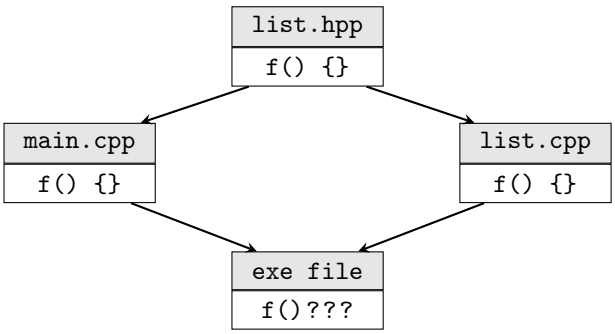
```
#include <iostream>

#include "list.hpp"
#include <iostream>

List::List(const List &other) : data(other.data), next(0)
{
    if (other.next != 0)
    {
        next = new List(*other.next);
    }
}

List& List::operator=(const List &other)
{
    if (this == &other)
        return *this;
    delete next;
    data = other.data;
    if (other.next)
    {
        next = new List(*other.next);
    }
    else
    {
        next = 0;
    }
    return *this;
}
```

```
void List::add(int data)
{
    if (next == 0)
    {
        next = new List(data);
    }
    else
    {
        next->add(data);
    }
}

Iterator Iterator::operator++()
{
    p = p->next;
    return *this;
}

int& Iterator::operator*() const
{
    return p->data;
}

ConstIterator ConstIterator::operator++()
{
    p = p->next;
    return *this;
}

int ConstIterator::operator*() const
{
    return p->data;
}
```

main.cpp:

```
#include <iostream>
#include "list.hpp"

void print(const List &l)
{
    for(ConstIterator it = l.begin(); it != begin(); ++it)
    {
        std::cout << *i << ' ';
    }
    std::cout << std::endl;
}

int main()
{
    List head(5);
    head.add(8);
    head.add(10);
    head.add(8);
}
```

Ez a szétválasztás sok egyéb előnnyel is jár: a `List`-hez tartozó információk sokkal kisebb helyen elférnek. Azonban ahogy a fenti megjegyzés is felhívta rá a figyelmet, a list.hpp továbbá is tartalmaz definíciókat! Ennek ellenére azt tapasztaljuk, hogyha több fordítási egységbe illesztjük be a headert, még akkor sem kapunk fordítási idejű hibát. Ennek a magyarázatához tegyünk egy kisebb kitérőt.



A `main.cpp`-ben vagy a `list.cpp`-ben lévő definíciója szerepeljen `f`-nek a futtatható fájlban?

19.1. Inline függvények

Tekintsük azt a példát, amikor a `void f() {}` függvényt is beillesztjük a headerbe: ha több fordítási egységet fordítanánk egyszerre, melybe ez a header be van illesztve, linkelési hibát kapnánk, mert `f` többször lesz definiálva. Ez azonban megkerülhető az `inline` kulcsszó használatával, segítségével ugyanis kiküszöbölhető a linker hiba: minden azonos nevű, visszatérési értékű, és paraméter listájú inline-ként definiált függvény definícióval együtt beilleszthető több különböző fordítási egységbe, és nem fog fordítási hibát okozni.
Ez úgy oldható meg, hogy a fordító a linkelés folyamán a definíciók közül egyet tetszőlegesen kiválasztást. Az osztályon belül kifejtett függvények implicit inline-ok, így sose okozhatnak fordítási hibát.

```
inline void f() {}
```

Az ábra jól demonstrálja a problémát. `f()` egy un. *strong reference*-el jön létre ha nem inline, így a linker hibát dob ha több fordítási egységben definiálva van. Ha azonban inline-ként adjuk meg, akkor *weak reference*-ként értelmezi, a meglevő definíciók közül tetszőlegesen kerül egy kiválasztásra. Ez nyilván azt is jelenti, hogy minden ilyen függvény definíciójának meg kell egyeznie, hisz kellemetlen meglepetés érhet minket, ha különböző definíciók közül olyat választ a fordító, melyre nem számítanánk (és ez egyben nem definiált viselkedés is).

19.1.1. Megjegyzés. A legtöbb fordítónál lehet egy LTO (*link time optimalization*) funkciót bekapcsolni, mely a linkelésnél optimalizál, többek között ott végzi el az inlineolást.

19.1.2. Megjegyzés. Az inline függvények hajlamosak erősen megnövelni a bináris kódot, így az erőltetett használatuk nem javallott.

19.1.3. Megjegyzés. Az inline kulcsszó egy javaslat a fordítónak, de nem parancs. Nem inline függvények lehetnek inline-ok, és inlineként definiált függvények lehet mégsem lesznek azok.

Azok a tagfüggvények, melyek nem az osztály törzsében vannak definiálva, nem lesznek inline-ok, ezért volt az, hogy mielőtt szétszedtük a listánkat header fájlra és fordítási egységre, linkelési hibát kaptunk (`Iterator` és `ConstIterator` pár tagfüggvénye külön volt véve).

Cseréljük le a print függvényt:

```
std::ostream& operator<<(std::ostream &os, const List &l)
{
    for(ConstIterator it = l.begin(); it != l.end(); ++it)
    {
        os << *it;
    }
    return os;
}
```

Sajnos ismét fordítási hibát kapunk, hisz a fordító nem tudja mi az az `ostream`, hisz az ismerős `iostream` könyvtár nincs include-olva. Ilyenkor azonban érdemes inkább az `iosfwd` headert beilleszteni az `iostream` helyett, mert ez minden beolvasással és kiíratással kapcsolatos osztály/függvénynek csak a deklarációját tartalmazza, és így csökken a fordítási egység mérete (azonban a cpp fájlban muszáj `iostream`-et használni, hogy a definíciók meglegyenek).

19.1.4. Megjegyzés. Ha szeretnénk egy `std::ostream` típusú objektumot használni, szükségünk lenne az `iostream` könyvtárra, hisz a fordítónak tudnia kéne, mekkora az `std::ostream` mérete, és ehhez szüksége van a teljes osztálydefinícióra. Azonban a referencia vagy pointer típusoknál erre nincs szükség, amíg nem akarunk egy tagfüggvényüket meghívni.

20. Névterek

A kódunkkal kapcsolatban felmerülhet egy másik probléma is: nagyon sok hasznos nevet elhasználtunk, pl. több `Iterator` nevű osztályt nem hozhatunk létre (az un. globális névtérbe vagy *global namespace*-be kerültek), különben a névütközés áldozatai leszünk. Pedig várhatóan nem csak ennek az egy konténernek szeretnék iterátort írni.

Megoldás lehet, hogyha inline class-t hozunk létre, azaz az iterátor teljes deklarációját beillesztjük a `List`-be, így csak a `List` lát rá `Iterator`-ra követlenül, mindenhol máshol úgy kell hivatkozni rá, hogy `List::Iterator`.

```
class List
{
public:
    class Iterator
    {
        //...
    };
    //...
};
```

Azonban szerencsésebb, ha az iterátorainkat egy névtérbe (*namesspace*) rakjuk.

```
namespace detail
{
    class Iterator
    {
        //...
    };
    class ConstIterator
    {
        //...
    };
}
```

Így az `Iterator` és `ConstIterator` osztályra a későbbiekben csak úgy hivatkozhatunk, ha megmondjuk, mely névtérből származnak. Ugyanezzel a módszerrel, ha létrehozunk pl. egy `Vector` nevű osztályt, annak is írhatunk egy `Iterator` nevű osztályt, amit pl. egy `VectorDetail` névtérbe tehetünk.

```
detail::Iterator it;
detail::ConstIterator cit;
```

A névterek segíthetnek abban, hogy logikai egységebre rendezzük a kódunkat. Az egyik legnagyobb ilyen egység az `std` névtér, mely tartalmaz minden függvényt, változót, stb, ami a standard részét alkotja.
Lehet névtereket egymásba is ágyazni, erre lehet példa a C++11-es `chrono` könyvtár, mely az `std` névteren belül számos dolgot a `chrono` alnévtérben tárol.

20.1. Typedef

A `typedef` kulcsszó szinonimák létrehozására használatos, és ha ügyesen használjuk, ki lehet használni valamennyi előnyét.

Hozzunk létre egy osztályt, melynek adatokat kell tárolnia. Tegyük fel, hogy egyenlőre nem fontos számunkra az, hogy milyen konténerben tároljuk az adatokat az osztályon belül, és a példa kedvéért követeljük meg ettől a leendő konténertől hogy rendelkezzen `push_back` tagfüggvénnyel. Az `std::vector` konténerrel ez így nézhetne ki:

```
class StoreIntData
{
private:
    std::vector<int> data;
};
```

Írjunk egy tagfüggvényt, mellyel két `StoreIntData` típus tárolt adatait össze tudjuk fűzni.

```
class StoreIntData
{
public:
    std::vector<int> merge(StoreIntData &other)
    {
        std::vector<int> ret;
        for(int i = 0; i<data.size(); i++)  ret.push_back(data[i]);
        for(int i = 0; i<other.size(); i++) ret.push_back(other.data[i]);
        return ret;
    }
private:
    std::vector<int> data;
};
```

20.1.1. Megjegyzés. Később látunk majd példát egy ennél jóval elegánsabb megoldásra is.

Tegyük fel, hogy az implementáció egy pontján úgy döntünk, mégse `std::vector`-ban, hanem `std::deque`-ban szeretnénk tárolni (erről a konténerről részletesen szó lesz később, egyenlőre legyen elég annyi, hogy az `std::vector`-hoz hasonló). Ekkor rákényszerülünk arra, hogy minden helyen, ahova `std::vector`-t írtunk, módosítanunk kelljen, és a kódismétlés áldozatai lettünk. Ehelyett használjunk egy `typedef`-et!

```
class StoreIntData
{
public:
    typedef std::deque<int> Container;

    Container merge(StoreIntData &other)
    {
        Container ret;
        for(int i = 0; i<data.size(); i++)  ret.push_back(data[i]);
        for(int i = 0; i<other.size(); i++) ret.push_back(other.data[i]);
        return ret;
    }
private:
    Container data;
};
```

Így ha módosítanunk kell a konténerünk típusát, elég a typedefet átírni.

Visszatérve a láncolt listánkra, a módosítás után `List` nem tudja, mi az az `Iterator`, hisz az egy `datail` nevű névtérben van, ezért vagy minden `Iterator`-t lecserélünk `detail::Iterator`-ra, vagy pedig létrehozunk egy szinonimát.

```
class List
{
public:
    typedef detail::Iterator Iterator;
    typedef detail::ConstIterator ConstIterator;
    //...
};
```

A typedef segítségével viszont még egy dolgot nyertünk: mivel ezen a szinonimák publikusak, így az osztályon kívül is tudunk rájuk hivatkozni így:

```
List::Iterator it = head.begin();
```

21. Nem template osztály átírása template osztályra

Már csak az a probléma, hogy `List` csak `int`-eket képes tárolni. Csináljunk belőle egy template osztályt! Feladatunk csupán annyi, hogy az osztály elé írjunk egy `template <typename T>`-t, és minden `List`-et `List<T>`-re, valamint minden `int`-et `T`-re cseréljünk. Ehhez nyilván az iterátorainkat is módosítani kell majd.

Időközben felmerül a hatékonyság kérdése is. A listánkban eddig mindent érték szerint vettünk át, ami `int`-nél (általában) hatékonyabb, mint a referencia szerinti, azonban template-eknél nem garantáljuk, hogy ilyen kis méretű típussal fogják példányosítani az osztályunkat, így ilyenkor célszerű úgy hozzáállni az implementáláshoz, hogy a leendő template paraméter egy nagyon nagy mérettel rendelkező típus lesz, melynél érték helyett hatékonyabb konstans referenciával visszatérni és átvenni a paramétereket.

21.0.1. Megjegyzés. Általában egy primitív típus, mint pl. az `int` vagy `char`, kisebb mérettel rendelkezik mint a hozzá tartozó pointer vagy referencia típus, így hatákonyabb ezeket a típusokat inkább érték szerint átvenni.

list.hpp:

```
#ifndef LIST_H
#define LIST_H

#include <iosfwd>

template<typename T>
class List;

namespace detail
{
    template<typename T>
    class Iterator
    {
    public:
        explicit Iterator(List<T> *p) : p(p) {}
        bool operator==(Iterator other) const { return p == other.p; }
        bool operator!=(Iterator other) const { return !(*this == other); }
        Iterator operator++();
        T& operator*() const;
    private:
        template<typename>
        friend class ConstIterator;
        List<T> *p;
    };

    template<typename T>
    class ConstIterator
    {
    public:
        ConstIterator(Iterator<T> it) : p(it.p) {}
        explicit ConstIterator(const List<T> *p) : p(p) {}
        bool operator==(ConstIterator other) const { return p == other.p; }
        bool operator!=(ConstIterator other) const
                                              { return !(*this == other); }
        ConstIterator operator++();
        const T& operator*() const; //konstans referenciával tér vissza!
    private:
        const List<T> *p;
    };
}

template <typename T>
class List
{
public:
    typedef detail::Iterator<T> Iterator;
    typedef detail::ConstIterator<T> ConstIterator;
```

```
    explicit List(const T &data_, List *next = 0) : data(data_), next(next)
        {}
    ~List() { delete next; }
    List(const List &other);
    List& operator=(const List &other);
    void add(const T &data);
    Iterator begin() { return Iterator(this); }
    ConstIterator begin() const { return ConstIterator(this); }
    Iterator end() { return Iterator(0); }
    ConstIterator end() const { return ConstIterator(0); }
private:
    friend Iterator;
    friend ConstIterator;
    T data;
    List *next;
};

template<typename T>
std::ostream &operator<<(std::ostream& os, const List<T> &l);

#endif
```

list.cpp:

```
#include "list.hpp"
#include <iostream>

template<typename T>
List<T>::List(const List &other) : data(other.data), next(0)
{
    if (other.next != 0)
    {
        next = new List(*other.next);
    }
}

template<typename T>
List<T> &List<T>::operator=(const List<T> &other)
{
    if (this == &other)
        return *this;
    delete next;
    data = other.data;
    if (other.next)
    {
        next = new List(*other.next);
    }
    else
    {
        next = 0;
    }
    return *this;
}

template <typename T>
void List<T>::add(const T &data)
{
    if (next == 0)
    {
```

```
        next = new List(data);
    }
    else
    {
        next->add(data);
    }
}

namespace detail
{
    template <typename T>
    Iterator<T> Iterator<T>::operator++()
    {
        p = p->next;
        return *this;
    }

    template <typename T>
    T& Iterator<T>::operator*() const
    {
        return p->data;
    }

    template <typename T>
    ConstIterator<T> ConstIterator<T>::operator++()
    {
        p = p->next;
        return *this;
    }

    template <typename T>
    const T& ConstIterator<T>::operator*() const
    {
        return p->data;
    }
}

template<typename T>
std::ostream& operator<<(std::ostream& os, const List<T> &l)
{
    for(typename List<T>::ConstIterator it = l.begin(); it != l.end(); ++it)
        //dependent scope!
    {
        os << *it << ' ';
    }
    os << std::endl;
    return os;
}
```

main.cpp

```
#include <iostream>
#include "list.hpp"

int main()
{
    List<int> head(5);
    head.add(8);
    head.add(10);
```

```
    head.add(8);
    std::cout << head;
}
```

Fordításnál azonban linkelési hibát kapunk, de miért? A list.hpp-ben benne van mindenféle deklaráció, és a list.cpp-ben meg több `List`-béli implementáció. A válasz a template-ek lustaságában rejlik.
Amikor a list.cpp-t ill. main.cpp-t fordítjuk, megfelelően létrejön az object fájl, mely tartalmazza példaképp azt, hogy a main függvény hivatkozik a `List<int>::add` függvényre. Linkeléskor a fordító keresi ennek a függvénynek az implementációját, azonban minden `List`-béli függvény template, és a list.cpp-ben semmit sem példányosítunk, az szinte teljesen üres lesz fordítás után.

Ennek következményeképp template osztályokat/függvényeket definícióval együtt a header fájlokban kell tárolni. Megoldás lehet, hogyha az egész list.cpp tartalmát bemásoljuk a list.hpp-be (itt már azonban muszáj lesz az `iosfwd`-t `iostream`-re váltani). Az átláthatóság azonban még így se esett áldozatul, mert a fájl tetején vannak a deklarációk, végén külön a definíciók, így még ugyanúgy könnyedén és gyorsan kinyerhető belőle a szükséges információ.

21.0.2. Megjegyzés. Ha nagyon fontosnak érezzük, hogy a definíciók külön fájlban legyenek, az is megoldható. Nevezzük át a list.cpp fájlt list_impl.hpp-ra, és include-oljuk a list.hpp végén.

22. Funktorok

Mielőtt belevetnénk magunkat az STL-ben lévő algoritmusokba és konténerekbe, fontos megismerkednünk a funktorokkal, melyek a rendezéseknél lesznek majd használatosak.

A funktor egy olyan osztály, melynek túl van terhelve a gömbölyű zárójel operátora (tehát kvázi meg lehet hívni).

Egy egyszerű példa:

```
struct S
{
    int x;
    int operator()(int y)
    {
        return x + y;
    }
};
int main()
{
    S s1, s2;
    s1.x = 5;
    s2.x = 8;
    std::cout << s1(2) << std::endl; //7
    std::cout << s2(2) << std::endl; //10
}
```

Bár `x` egy objektum, függvényként is funkcionál. A funktorok segítségével nagyon könnyedén tudunk objektumokat rendezni. Tekintsünk is erre egy példát!

22.1. Predikátumok

Írjunk egy funktort, mely egész számok számokat tud összehasonlítani érték szerint.

```
template <class Compr>
bool f(int a, int b, Compr c)
{
    return c(a, b);
}

struct Less
{
```

```
    bool operator()(int a, int b) const
    {
        return a < b;
    }
};

int main()
{
    if( f(2,3, Less()) )
        std::cout << "2 kisebb mint 3! ";
}
```

Kimenet: `2 kisebb mint 3!`
A fenti funktor (`Less`) egy un. **bináris predikátum** (*binary predicate*), azaz a gömbölyű zárójel operátora két azonos típusú objektumhoz rendel egy logikai értéket (matamatikai nyelven $Less : T \times T \to \mathbb{L}$, ahol T ismert típus). Figyeljük meg, hogy a harmadik paraméter egy névtelen temporális változó.

22.1.1. Megjegyzés. Figyeljük meg azt is, hogy ebben az esetben érték szerint vettük át a hamradik paramétert: ez nem csak hatékonyabb, hisz `Less` mérete minimális (nincs adattagja), de mivel `Less()` egy jobbérték, csak így, vagy konstans referenciával tudnánk átvenni.

Amennyiben egy `Less` típusú objektum gömbölyű zárójel operátorát 2 `int`-el meghívjuk, és a visszatérési érték `true`, akkor az első szám a kisebb, ellenkező esetben a második. Ez alapján az egyenlőség is levezethető: ha egy a szám nem nagyobb b-nél, és b sem nagyobb a-nál, akkor egyenlőek.

$$a = b \quad \Leftrightarrow \quad \neg(a < b) \wedge \neg(b < a)$$

```
int main()
{
    int a = 2, b = 2;
    if( f(a, b, Less()) == false && f(b, a, Less()) == false )
        std::cout << "2 és 2 egyenlő!";
}
```

Kimenet: `2 és 2 egyenlő!`
Könnyű látni, hogy általánosan beszélve, `Compr` összehasonlító funktorral a és b `T` típusú objektumoknál

$$a \text{ és } b \text{ ekvivalens} \quad \Leftrightarrow \quad \neg\, Compr(a,b) \wedge \neg\, Compr(b,a).$$

Azokat a funktorokat, melyeknek az `()` operátora csak **egy** adott típusú objektumot várnak és `bool` a visszatérési értékük, *unary predicate*-nek hívjuk. Írassunk ki egy tömb páros elemeit!

```
template <class T, class Pred> //pred mint predicate
void printIf(T *start, int size, Pred pred)
{
    for (int i = 0; i<size; i++)
    {
        if( pred(start[i]) )
            std::cout << start[i] << std::endl;
    }
}

struct IsEven //unary predicate
{
    bool operator()(const int &a) const
    {
        return a % 2 == 0;
    }
};

int main()
```

```
{
    int t[] = {1,2,3,4,5,6};
    printIf(t, sizeof(t)/sizeof(t[0]), IsEven()); //2 4 6
}
```

22.1.2. Megjegyzés. Miért használunk funktorokat függvénypointerek helyett? Világos, hogyha egyedi rendezést szeretnénk, akkor muszáj ezt az információt valahogy átadni. A funktorok erre alkalmasabbak, példaképp vegyük ehhez egy olyan template függvényt, melynek egyik template paramétere egy olyan funktort vár, melynek `()` operátora egy paramétert vár és `bool`-al tér vissza, és feladata az erre igazat adó elem megkeresése.
Hogyan tudnánk mi funktorok helyett függvénypointerrel a második páros számot megkeresni vele? Vagy az ötödik 8-al oszthatót? Nos, függvényekkel igencsak nehezen (kb statikus adattagokra kényszerülnénk, vagy egy hasonlóan nem épp elegáns megoldásra), azonban egy funktorban létrehozhatunk egy számlálót, melyet tudunk növelgetni.

23. STL konténerek

Az *STL* a *Standard Template Library* rövidítése.

23.1. Bevezető a konténerekhez

C++ban az egyetlen tároló alapértelmezetten a tömb. Azonban az meglehetősen kényelmetlen: a tömb egymás mellett lévő memóriacímek összessége, nem tehetjük meg azt, hogy csak úgy hozzáveszünk 1-1 elemet (mi van ha valaki más már írt oda?). Ezért jobban járunk, ha vagy írunk egy egyedi konténert (pl. az általunk létrehozott `List`), vagy pedig válogatunk az előre megírt STL konténerek között.
Három konténertípust különböztetünk meg:

- **Szekvenciális**: Olyan sorrendben tárolja az adattagokat, ahogyan betesszük őket.
  Példa: `std::vector`, `std::deque, std::list.`
- **Asszociatív**: Az elemek rendezettek a konténerben.
  Példa: `std::set`, `std::map`, `std::multiset`, `std::multimap`.
- **Konténer adapter**: Egy meglévő konténert alakítanak át, általában szogorítanak. Példaképp ha vesszük az `std::deque` konténert, ami egy kétvégű sor, könnyen tudunk belőle egy egyvégű sort csinálni. Ezen a konténerek általában egy másik konténert tárolnak, és annak a funkcióit szigorítják.
  Példa: `std::queue`, `std::stack`.

Minden STL konténer rendelkezik konstans és nem konstans iterátorral, fordított irányú és konstans fordított irányú iterátorral melyre így hivatkozhatunk:

Iterátor típus	Ahogy hivatkozhatunk rájuk	Első elem	Past-the-end
iterátor	`/*konténer név*/::iterator`	`begin()`	`end()`
konstans iterátor	`/*konténer név*/::const_iterator`	`cbegin()`	`cend()`
fordított irányú iterátor	`/*konténer név*/::reverse_iterator`	`rbegin()`	`rend()`
fordított irányú konstans iterátor	`/*konténer név*/::reverse_const_iterator`	`crbegin()`	`crend()`

A következő leírásban nem fogunk minden létező tagfüggvénnyel foglalkozni: egyrészt olyan sok van, hogy azt teljesen irreális észben tartani, másrészt sok ilyenhez érdemi hozzáfűznivalót én nehezen tudnék tenni, így az egyes szekciók végén található link az adott konténerhez.

23.1.1. Megjegyzés. Nagyon fontos képesség az is, hogy valaki hogyan tud utánanézni valaminek, amivel nincs teljesen tisztában, így erősen javallott a `cppreference.com`-el való ismerkedés, illetve más helyeken lévő információk böngészése is. A könyvben található linkek túlnyomótöbbségben kattinthatóak, és érdemes is ellátogatnunk ezen oldalakra.

23.2. vector

A `<vector>` könyvtárban található, maga a konténer az `std::vector` névre hallgat. Előzménytárgyakból ismerős lehet ez a konténer, más néven mint dinamikus tömb hivatkoztunk rá. Bár a pontos definíciója mint megannyi STL-béli algoritmus és konténer implementációfüggő, vannak olyan tulajdonságok, melyeket elvár a szabvány: pl. rendelkezzen `push_back` függvénnyel, a `[]` operator műveletigénye legyen konstans, stb.

Tekintsünk az `std::vector` pár alkalmazását.

```
#include <vector>
#inclide <iostream>

int main()
{
    std::vector<int> v;

    //elem beszúrása
    for (int i = 0; i<11; i++)
        v.push_back(i);

    //utolsó elem törlése
    v.pop_back();

    //végigiterálás a konténeren a már jól ismert módon
    for (int i = 0; i<v.size; i++)
        std::cout << v[i] << std::endl; // 0 1 2 3 4 5 6 7 8 9

    //iterátorok használata
    for (std::vector<int>::iterator it = v.begin(); it != v.end(); ++it)
        std::cout << *it << std::endl; // 0 1 2 3 4 5 6 7 8 9
}
```

A `vector` leggyakrabban dinamikusan lefoglalt tömbben tárolja az adatainkat, viszont – ahogy az korábban is említve volt – egy tömb mérete nem növelhető. Ezt az `std::vector` a következőféleképpen oldja meg: ha több elemet szeretnénk beszúrni, mint amennyi az adott `vector` kapacitása, akkor lefoglal egy nagyobb memóriaterületet (leggyakrabban kétszer akkorát), és minden elemet átmásol erre az új memóriaterületre.
Így a `push_back`-nek a műveletigénye amortizált konstans: Általában konstans, de ha új memóriaterületet kell lefoglalni és a meglevő elemet átmásolni, akkor lineáris.

23.2.1. Megjegyzés. Számos okból a `vector` a leggyakoribb választás, ha konténerre van szükségünk. Flexibilitása és gyorsasága kiemelkedő, azonban megvannak a maga gyenge pontjai: a konténer közepére elemet beszúrni például csak úgy tudunk, ha egyesével valamennyi elemet odébb másolunk.

Feltűnhet, hogy ezek a műveletek, csakúgy mint számos egyéb melyet a cppreference-n olvashatunk, a konténer végére fókuszál, sőt, az elejére vonatkozó műveletek nincsenek is implementálva (nincs semmilyen `push_front` vagy `pop_front`). Ennek az az oka, hogy az `std::vector`-nál a konténer végének a módosítása a leghatékonyabb, ha a közepén/elején szeretnénk módosító művelteket végrehajtani, az gyakran a környező elemek odébb másolásával jár.

Az `std::vector` tagfüggvényei, mint a nemsoká következő STL algoritmusok, többnyire iterátorokat várnak paraméterül. A következő példában töröljük ki a 4. elemet, majd szúrjuk is vissza!

```
int main()
{
    std::vector<int> v;
    for (int i = 0; i<10; i++)
        v.push_back(i);

    for (int i = 0; i<v.size(); i++)
        std::cout << v[i] << ' '; // 0 1 2 3 4 5 6 7 8 9

    std::vector<int>::iterator it = v.begin() + 3;
```

```
    v.erase(it);
    for (int i = 0; i<v.size(); i++)
        std::cout << v[i] << ' '; // 0 1 2 4 5 6 7 8 9

    it = v.begin() + 3;
    v.insert(it, 3);
    for (int i = 0; i<v.size(); i++)
        std::cout << v[i] << ' '; // 0 1 2 3 4 5 6 7 8 9
}
```

Az iterátorok kezelésének azonban komoly veszélyei is vannak, a legnagyobb gonosz itt az un. iterátor invalidáció (*iterator invalidation*). Amikor a `vector` által lefoglalt dinamikus tömb mérete túl kicsi az újabb elemek beszúrásához, és egy újabb tömbbe másolja át őket, minden iterátor, pointer és referencia ami a régebbi tömbre hivatkozott invalidálódik, lévén olyan területre hivatkoznak, melyeket már felszabadultak.

Ugyanígy a konténer közepéről történő törlés, vagy a konténer közepére történő beszúrás is iterátor invalidációval jár.

Az invalidálódott objektumokkal végzett műveletek nem definiált viselkedést eredményeznek.

```
int main()
{
    std::vector<int> v;
    v.push_back(3);
    std::vector<int>::iterator it = v.begin();

    for (int i = 0; i<1000; i++)
        v.push_back(i);

    *it = 10; // nem definiált viselkedés
}
```

Az invalidáció elkerülése végett számos tagfüggvény egy iterátorral tér vissza, erre lehet példa az `insert`, mely az új elemre, vagy az `erase`, mely az utolsó eltávolított elem rákövetkezőjére hivatkozik.

Töröljünk egy vektorból minden páratlan számot!

```
int main()
{
    std::vector<int> v;
    for (int i = 0; i<10; i++)
        v.push_back(i);
    for(std::vector<int>::iterator it = v.begin(); it != v.end(); )
    {
        if(*it % 2 == 1)
            it = v.erase(it); //iterátor invalidáció elkerülése végett
        else
            ++it;
    }
}
```

23.2.2. Megjegyzés. Oldjuk meg, hogy a fenti 4-es értékű elem törlésénél `it`-nek ne újra `v.begin() + 3`-at adjunk értékül, hanem használjunk ki hogy az `erase` függvény visszatér egy iterátorral!

Link: `http://en.cppreference.com/w/cpp/container/vector`.

23.2.3. Megjegyzés. Az `std::deque` működése az eddiegiek alapján triviális, így annak megismerését az olvasóra bízom.

23.3. set

Az `std::set` a `<set>` könyvtárban található. Ez a konténer a matematikai halmazt valósítja meg: egyedi elemeket tárol, így ha egy olyan elemet próbálnánk beszúrni, mellyel ekvivalens már szerepel a `set`-ben, nem történne semmi (*no-op*).

A szabvány azt is megköveteli, hogy ez a konténer rendezett legyen. Ahhoz, hogy ezt meg tudja valósítani, szüksége van a konténernek egy bináris predikátum funktorra is mint template paraméter a tárolandó `T` típus mellett.

Ez utóbbi template paramétert nem kell feltétlenül megadni, alapértelmezetten ugyanis az `std::set` az `std::less<T>` alapján rendez, mely gyakorlatilag az `<` operátorral ekvivalens.

Ezt a következőféleképpen képzelhetjük el:

```
namespace std
{
    template <class T>
    struct less
    {
        bool operator()(const T &lhs, const T &rhs) const
        {
            return lhs < rhs;
        }
    };

    template <class T, class Compr = less<T> >
    class set;
}
```

23.3.1. Megjegyzés. cppreference-en megfigyelhető, hogy ennél több template paraméterrel is rendelkezik az `std::set` – mivel ezek mind rendelkeznek alapértelmezett értékkel, a továbbiakban az egyszerűség kedvéért figyelmen kívül hagyjuk őket.

23.3.2. Megjegyzés. Az `std::set` általában piros-fekete bináris faként van implementálva, a hatékonyság végett. Ez (vagy egy hasonlóan hatékony implementáció) fontos, mert a szabvány elvárja, hogy az `insert`, és sok egyéb tagfüggvény műveletigénye logaritmikus legyen.

Tekintsünk pár példát az `std::set` alkalmazására! Mivel ez a konténer rendezett, így ha sok adatot pakolunk bele, nagyon nehéz megmondani, hogy az elemek milyen sorrendben lesznek, így az `std::set` elemeihez többnyire csak iterátorokkal tudunk hozzáférni.
Lássunk példát pár speciális tagfüggvényre is!

```
#include <set>
#include <iostream>

template <class T>
void printSet(const std::set<T> &s)
{
    for(typename std::set<T>::const_iterator it = s.begin();
                                                    it != s.end(); ++it)
        std::cout << *it << ' ';
}

int main()
{
    std::set<int> si;
    for(int i = 10; i>0; i--) //forditva!
        si.insert(i);
    std::cout << si.size() << std::endl; // 10
    si.insert(5); //5 már szerepel a halmazban
    std::cout << si.size() << std::endl; // 10

    printSet(si); // 1 2 3 4 5 6 7 8 9 10
    //operator< növekvően rendez, hiába fordított sorrendben illesztettük be
        az elmeket

    //adott értékű elem törlése
```

```
    std::set<int>::iterator it = si.find(4);
    if(it != si.end())
        si.erase(it);
    printSet(si); // 1 2 3 5 6 7 8 9 10

    //töröljük ki az [6, 8) intervallumot!
    std::set<int>::iterator begin = si.find(6), end = si.find(8);
    if(begin != si.end())
        si.erase(begin, end);
    printSet(si); // 1 2 3 5 8 9

}
```

Figyeljük meg, hogy az elemek törlésekor, leellenőriztük, valóban szerepel-e az az adott elem a halmazban. Mivel a `find` tagfüggvény egy past-the-end iterátorral tér vissza, hogyha az adott elemet nem találja, az azzal való egyenlőség segítéségvel tudjuk ezt megvizsgálni.
Ha egy past-the-end iterátort próbálunk törölni az `erase` függvénnyel, nem definiált viselkedést kapunk.

A következő példában tekintsünk egy `int`-től különböző típust tároló halmazt. Ne feledjük, mivel e konténer rendezi az elemeit, mindenképpen muszáj a struktúránk mellé még valamit megírnunk. Ez lehet az alapértelmezetten alkalmazott `operator<` függvény vagy egy új funktor.

```
struct Point
{
    int x, y;
    Point(int _x, int _y) : x(_x), y(_y) {}
};

std::ostream& operator<<(std::ostream& os, const Point &p)
{
    os << p.x << ' ' << p.y;
    return os;
}

//operator< y koordináta szerint rendez
bool operator<(const Point &lhs, const Point &rhs)
{
    return lhs.y < rhs.y;
}

//LessByX funktor x koordináta szerint
struct LessByX
{
    bool operator()(const Point &lhs, const Point &rhs) const
    {
        return lhs.x < rhs.x;
    }
};

int main()
{
    std::set<Point> spy;
    spy.insert(Point(3, 1));
    spy.insert(Point(1, 3));
    spy.insert(Point(2, 2));
    for(std::set<Point>::iterator it = spy.begin(); it != spy.end(); ++it)
        std::cout << *it << ", "; // 3 1, 2 2, 1 3,

    std::set<Point, LessByX> spx;
    spx.insert(Point(3, 1));
```

```
    spx.insert(Point(1, 3));
    spx.insert(Point(2, 2));
    for(std::set<Point, LessByX>::iterator it = spx.begin();
                                                    it != spx.end(); ++it)
        std::cout << *it << ", "; // 1 3, 2 2, 3 1,

    spy.insert(Point(1, 1)); spx.insert(Point(1, 1));
    std::cout << spy.size() << ' ' << spx.size() << std::endl; // 3 3
}
```

23.3.3. Megjegyzés. A funktorok mellett az beszúrófüggvények használata is jó példa arra, mikor hasznosak a név nélküli temporális változók.

A fenti példában megfigyelhető, hogy bár 1, 1 koordinátákkal rendelkező pontot egyik halmaz sem tartalmaz, az mégis **ekvivalens** egy, a halmazokban már szereplő elemekkel.

$$\begin{aligned} \texttt{spy} \text{ esetében:} \quad & \neg\big(\texttt{Point}(1,1) < \texttt{Point}(3,1)\big) \quad \wedge \quad \neg\big(\texttt{Point}(3,1) < \texttt{Point}(1,1)\big) \\ \texttt{spx} \text{ esetében:} \quad & \neg\big(\texttt{Point}(1,1) < \texttt{Point}(1,3)\big) \quad \wedge \quad \neg\big(\texttt{Point}(1,3) < \texttt{Point}(1,1)\big) \end{aligned}$$

Vagy másféleképpen a funktorok neveivel:

$$\begin{aligned} \texttt{spy} \text{ esetében:} \quad & \neg\texttt{less}\big(\texttt{Point}(1,1), \texttt{Point}(3,1)\big) \quad \wedge \quad \neg\texttt{less}\big(\texttt{Point}(3,1), \texttt{Point}(1,1)\big) \\ \texttt{spx} \text{ esetében:} \quad & \neg\texttt{LessByX}\big(\texttt{Point}(1,1), \texttt{Point}(1,3)\big) \quad \wedge \quad \neg\texttt{LessByX}\big(\texttt{Point}(1,3), \texttt{Point}(1,1)\big) \end{aligned}$$

Egy egyedi rendezés használata veszélyekkel is járhat azonban. Térjünk át `std::string`-ekre, és rendezzünk azok hossza szerint.

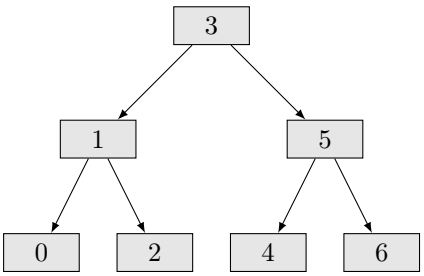
```
struct strlen
{
    bool operator()(const std::string &lhs, const std::string &rhs) const
    {
        return lhs.length() < rhs.length();
    }
};

int main()
{
    std::set<std::string, strlen> s;
    s.insert("C++");
    s.insert("Java");
    s.insert("Haskell");
    s.insert("GOD");
    std::cout << s.size() << std::endl; // 3
    std::cout << s.count("GOD") << ' ' << s.count("ADA"); // 1 1
}
```

Az eddig leírtak alapján látható, hogy az `insert` függvény nem teszi be a `GOD`-ot, lévén az a rendezés szerint a `C++`al ekvivalens. Azonban a rendezésünknek egy kellemetlen hátulütője az, hogy annak ellenére, hogy `GOD` nem került beszúrásra, ha rákérdezünk hány `GOD`-al ekvivalens elemet tartalmaz a halmaz, mégis 1-et kapunk, sőt, minden 3 hosszú `string`-re.

23.3.4. Megjegyzés. Könnyű megállapítani, hogy a fenti rendezés szimmetrikus, reflexív és tranzitív, így ekvivalenciaosztályokra osztja a `string`-ek halmazát hossz szerint.

Most figyeljük meg (emlékezzünk vissza dimatra), mi történik hogyha szigorú rendezés helyett egy gyenge rendezést definiálunk! Miért lehet ez problémás?

```
struct strlenWrong
{
    bool operator()(const std::string &lhs, const std::string &rhs) const
    {
        return lhs.length() <= rhs.length(); //ekvivalensek is lehetnek
```



Az `std::set` egy lehetséges ábrázolása, bináris keresőfával. Erre az irányított gráfra igaz, hogy minden gyökér a bal oldali gyerekénél nagyobb, és a jobb oldali gyerekénél kisebb.

```
    }
};

int main()
{
    std::set<std::string, strlenWrong> s;
    s.insert("C++");
    s.insert("Java");
    s.insert("Haskell");
    s.insert("GOD");
    std::cout << s.count("GOD") << std::endl; // 0
    std::cout << s.size() << std::endl; // 4
}
```

Keressünk magyarázatot erre az eredményre, vizsgáljuk meg `C++` és `GOD` ekvivalenciáját.

$$\neg\,\texttt{strlenWrong}(\texttt{"GOD"}, \texttt{"C++"}) \quad \wedge \quad \neg\,\texttt{strlenWrong}(\texttt{"C++"}, \texttt{"GOD"})$$

Látjuk, hogy ez a formula hamisat ad, így nem bizonyulnak majd ekvivalensnek. Ha nem szigorú részben rendezést használunk, hanem gyengét, akkor a reflexivitást is elvesztjük. Ez azt jelenti, hogy egy elem nem lehet ekvivalens önmagával!
Így `strlenWrong` nem egy jó rendezés, mivel ha ekvivalenciát vizsgálunk, sose fog igazat adni, sose tudjuk meg, egy adott benne van-e, és az elemek törlése is sok problémához vezetne.

23.3.5. Megjegyzés. Bár sokszor volt ez fentebb leírva, hangsúlyozandó hogy az `std::set` **nem** az `==` operátor segítségével vizsgálja az ekvivalencát. Két objektum lehet egyszerre **ekvivalens** és nem **egyenlő**, függően az `==` operátor és a rendezés implementációjától.

Link: `http://en.cppreference.com/w/cpp/container/set`

23.3.6. Megjegyzés. Az `std::multiset` működése az eddigiek alapján triviális, így annak megismerését ismét az olvasóra bízom.

23.4. list

Az `std::list` konténer a `<list>` könyvtár része, és nagyon hasonló ahhoz, mint amit mi írtunk, viszont létezik fejelemmel, és kétirányú.

Az `std::vector`-ral szemben ennek a konténernek nincs `[]` operátora, hiszen mielőtt egy adott indexű elemet vissza tudna adni, el kell oda lépegetnie egyesével. Szögletes zárójel operátort csak akkor szokás írni, hogy ha az nagyon hatékony, lehetőleg konstans műveletigényű, de ez a listánál nem teljesül.

23.4.1. Megjegyzés. Bár elméletben gyorsabb egy láncolt listába beszúrni a `vector`-ral (vagy hasonló konténerrel) szemben, hisz csak pár pointert kell átállítani, ez a gyakorlatban csak kivételes esetekben teljesül. Ennek az az oka, hogy bár az elemeket egyesével odébb kell tolni egy vektorban, azok szekvenciálisan vannak a memóriában, és mivel a processzor számít arra, hogy nem csak egy adott elemet, de a környezőket is módosítani szeretnénk, minden alkalommal amikor egy adott elemet kérdezünk le, azzal együtt a környező elemeket is visszaadja nekünk. Ennek következtében kevesebb processzorművelettel végrehajtható a tologatás.
Mivel egy láncolt listában egyesével kell haladni az egyik elemről a következőre, ez gyakran annyira költséges, hogy jobban járunk a vektor alkalmazásával akkor is, ha sokat kell a konténer közepére beszúrni.

Kivételes eset lehet, ha nagyon sok elemből álló listánk van, és azok nagy méretű objektumokat tárolnak, valamint nagyon gyakran kell két elem közé beszúrni. Ilyenkor valóban hatékonyabb tud lenni a `list`.

Mivel már a korábban megismerkedtünk a láncolt listákkal, valamint a korábbi szekciókban mutatott tagfüggvények alkalmazása gyakran teljesen megegyezik a `list`-nél használatosakkal, túl komoly példákat itt nem fogunk venni.
Azonban érdemes felhívni a figyelmet egy problémára. Kérdezzük le egy lista harmadik elemét!

```
#include <iostream>
#include <list>

int main()
{
    std::list<int> l;
    for (int i = 0; i < 5; i++)
        l.push_back(i);
    std::list<int>::iterator it = l.begin();
    ++it; ++it;
    std::cout << *it << std::endl; // 2
}
```

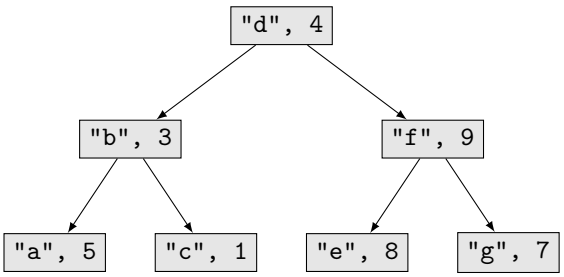
Az iterátorok léptetésére kell hogy legyen egy egyszerűbb módszer. Írhatnánk egy ciklust, azonban beszédesebb lenne egy léptető függvény írása.
Azonban nem ártana, ha nem csak tetszőleges típust tároló lista iterátorát, hanem tetszőleges konténer iterátorát is tudná ez a függvény léptetni. Mivel az `std::vector`-nál bármikor ugorhatunk egy tetszőleges elemre, hatékonyabb lenne annak konstans műveletigényű léptető függvényt írni: jó lenne egy olyan algoritmust találni, mely e kettőt egybefoglalja (erre később látunk is majd példát az STL algoritmusoknál).

Ha böngészünk cppreference-en, feltűnhet, hogy az `std::vector`-ral szemben létezik `push_front` és `pop_front` metódus: ezeket az `std::vector`-ral ellentétben konstans idő alatt el lehet végezni (elegendő a fejelemet a rákövetkező elemre állítani és kész is).

Léteznek kifejezetten a listára specifikus műveletek is. Ezek azért nagyon fontosak, mert nem igénylik azt, hogy az adott objektumot lehessen másolni (vagy mozgatni, C++11 szerint). Ez olyankor is fontos lehet, ha például a tárolt objektumok mérete nagyon nagy, és költséges lenne másolni. Példaképp, a `splice` tagfüggvény egy másik listát (is) vár paraméterül, és azt a listát fűzi be a saját elemei közé.

```
std::ostream& operator<<(std::ostream& out, const std::list<int> &l)
{
    for (std::list<int>::const_iterator it = l.begin(); it != l.end(); it++)
    out << *it << ' ';
    return out;
}

int main()
{
    std::list<int> list1, list2{};
    for(int i = 0; i<5; i++)
        list1.push_back(i);
    for(int i = 0; i<5; i++)
        list2.push_front(i);
    std::cout << list1 << std::endl; // 0 1 2 3 4
    std::cout << list2 << std::endl; // 4 3 2 1 0

    list1.splice(list1.end(), list2);
    std::cout << list1 << std::endl; // 0 1 2 3 4 4 3 2 1 0
}
```

Link: `http://en.cppreference.com/w/cpp/container/list`



Az `std::map` egy lehetséges ábrázolása, bináris keresőfával. Az ábrán egy `std::string, int` párosokat tároló `map` szerepel – figyeljük meg, hogy a a konténer a kulcs szerint rendez, az értékek a rendezés szempontjából nem számítanak.

23.5. map

Az `std::map` a `<map>` könyvtárban található. Ez egy egyedi kulcs-érték párokat tároló konténer, mely minden kulcshoz egy értéket rendel. Működése nagyon hasonlít az `std::set`-ére, annyi különbséggel, hogy párokat tárol (így két kötelező template paramétere van), és mindig a kulcs szerint rendez.

Van egy speciális beszúró függvénye is:

```
#include <map>
#include <iostream>

int main()
{
    std::map<std::string, int> m;
    m["Hello"] = 42;
    m["xyz"] = 8;
    m["Hello"] = 9;
    std::cout << m.size() << std::endl; // 2
    std::cout << m["Hello"] << std::endl; // 9
}
```

A `[]` operátor egy új kulcsot hoz létre, és ahhoz rendel egy új értéket. Az utolsó sornál mivel a `"Hello"`-t már tartalmazza a map, így annak az értékét felülírja.

Sajnos ez az operátor azonban okozhat pár kellemetlen meglepetést is.

```
int main()
{
    std::map<std::string, int> m;
    std::cout << m.size() << std::endl; // 0
    if(m["c++"] != 0)
        std::cout << "nem 0" << std::endl;
    std::cout << m.size() << std::endl; // 1
}
```

A `[]` operátor úgy működik, hogy ha a `map` nem tartalmazza a paraméterként kapott kulcsot, akkor létrehozza és beszúrja, ha benne van, visszaadja az értékét. Bár mi nem tettük bele azt hogy `c++` szándékosan, de pusztán azzal, hogy rákérdeztünk, akaratlanul is megtettük. (Figyeljük meg, hogy akkor tudnánk új elemet hozzátenni, ha azt írnánk pl. hogy `m["C++"] = 3`; azonban mi nem adtunk meg ehhez a kulcshoz értéket. Ilyenkor a map alapértelmezett értéket rendel a kulcshoz, szám típusoknál 0-t, így be fog szúrni egy `("c++", 0)` párt.)

```
bool contains(const std::map<std::string, int> &m)
{
    //m["c++"]; fordítási hiba: [] operator nem konstans függvény.
    return m.find("C++") != m.end();
}
```

Ez a függvény helyesen vizsgálja, hogy a `c++` benne van-e. (Ha cppreference-en rákeresünk, a `map`-nek a `find` nevű tagfüggvénye egy iterátort ad vissza a talált elemre, illetve visszaadja egy past-the-end iterátort ha nem találja meg.)

`std::map`-ban az iterálás kicsit trükkösebb, ugyanis `std::map` párokat tárol, méghozzá egy más alapból megírt struktúrát használ, az `std::pair`-t, ami kb. így néz ki:

```
template <typename T1, typename T2>
struct pair
{
    T1 first;
    T2 second;
};
```

Így az az adott kulcs ill. érték lekérdezése így fog kinézni:

```
for(std::map<std::string, int>::iterator i = m.begin(); i != m.end(); i++)
{
    std::cout << i->first << " " << i->second;
}
```

Amennyiben új elemeket szúrunk be, gyakran megkerülhető a problémás `[]` használata. Az `std::map` insert függvénye egy `std::pair`-t vár paraméterül, melyet könnyen tudunk alkotni `std::make_pair` függvény segítségével.

```
std::map<int, char> m;
m.insert(std::make_pair(1, 'a'));
```

Link: `http://en.cppreference.com/w/cpp/container/map`

23.5.1. Megjegyzés. Az `std::multimap` ehhez szintén hasonló, így gyakorlásként ezt ismét az olvasóra bíznám.

24. Iterátor kategóriák

Korábban már elhangzott, hogy az általunk implementált `List`-hez tartozó `Iterator` és `ConstIterator` un. forward iterátorok. A forward iterator egy iterátor kategória – ebből több is van, és elengedhetetlenek az STL algoritmusok megértéséhez. 4 iterátor típust különböztetünk meg: input iterator, forward iterator, bidirectional iterator és random access iterator.

- Az **input iterator**-okon legalább egyszer végig lehet menni egy irányba, továbbá rendelkeznek `++`, `*`, `==` és `!=` operátorral.
- A **forward iterator**-okon többször is végig lehet menni, de csak egy irányba. Továbbá rendelkeznek `++`, `*`, `==` és `!=` operátorral.
- A **bidirectional iterator**-okon többször is végig lehet menni, mindkét irányba. Továbbá rendelkeznek `++`, `--`, `*`, `==` és `!=` operátorral.
- A **random access iterator**-okon többször is végig lehet menni, mindkét irányba, ezen felül a két iterátor között bármelyik elemre azonnal lehet hivatkozni. Továbbá rendelkeznek `++`, `--`, `*`, `==`, `!=`, – (két iterátor távolságát adja meg), összeadás `int`-el (paraméterként kapott mennyiségű elemmel előrelépés), kivonás `int`-el (hátralépés) operátorral/függvénnyel.

E listán látható, hogy pl. egy random access iterator sokkal flexibilisebb, és gyorsabb is, hisz bármikor bármelyik elemhez hozzáférhetünk, míg a mi listánk forward iteratora sokkal limitáltabb.

Az STL konténerek iterátorai is különböző iterator kategóriákban vannak:

STL konténer	Iteratorának kategóriája
`std::vector` `std::deque`	`random access iterator`

`std::list`
`std::set`
`std::multiset`
`std::map`
`std::multimap`
	`bidirectional iterator`

24.0.1. Megjegyzés. `input iterator`-ra példa lehet az `std::istream_iterator` osztály, míg az általunk megírt lista iterátora egy `forward_iterator`-nak számít.

Világos, hogy egy flexibilisebb iterátorral több mindent meg tudunk tenni, vagy ugyanazt a funkciót hatékonyabban is meg tudjuk valósítani. Emiatt minden STL algoritmusnak (mint pl. az `std::find`) tudnia kell a template paraméterként kapott iterátor kategóriáját.

Egy iterátornak úgy tudjuk a legegyszerűbben megadni a típusát, ha származunk az `std::iterator` típusból.

24.0.2. Megjegyzés. Az öröklődés később lesz alaposabban boncolgatva, egyenlőre mondjuk azt, hogy az öröklődés következtében mindent átpakolunk az `std::iterator` osztályból a mi iterátorunkba.

Link: `http://en.cppreference.com/w/cpp/iterator/iterator`

Látható hogy ennek osztálynak két kötelező template paramétere van, egy iterátor típus tag, és dereferáló operátor visszatérési értéke.

```
template <class T>
class Iterator : public std::iterator<std::forward_iterator_tag, T>
{
    //..
};

template <class T>
class ConstIterator : public std::iterator<std::forward_iterator_tag, T>
{
    //..
};
```

A cppreference-en megfigyelhető, hogy az `std::iterator` több típust is tartalmaz, mely jelöli hogy az iterátorunk milyen kategóriában van, milyen típussal tér vissza a dereferáló operátor, stb. Ennek segítségével pl. a listánk iterátorának kategóriája lekérdezhető az `List::Iterator::iterator_category`-val (erre nemsokára példát is látni fogunk).

24.0.3. Megjegyzés. Ennek alaposabb megértését az olvasóra bízom.

Így már fogjuk tudnuk majd használni az `STL` algoritmusokat is a konténereinken.

25. STL algoritmusok

25.1. Bevezető az algortmusokhoz

Az STL algoritmusok az `<algorithm>` könyvtárban találhatóak, és számos jól ismert és fontos függvényt foglalnak magukba, mint pl. adott tulajdonságú elem keresése, partícionálás, szimmetrikus differencia meghatározása, stb.

Az STL algoritmusok alap gondolatmenetét jól demonstrálja az alábbi példa: Írjunk egy függvény mely egy `std::vector<int>` konténernek a második adott értékű elemére hivatkozó iterátort ad vissza!

```
#include <vector>
#include <iostream>

typedef std::vector<int>::const_iterator VectIt;

VectIt findSecond(VectIt begin, VectIt end, const int &v)
{
    while(begin != end && *begin != v)
    {
        ++begin;
    }
    if (begin == end)
        return end;
    ++begin;
```

```
    while(begin != end && *begin != v)
    {
        ++begin;
    }
    return begin;
}

int main()
{
    std::vector<int> v;
    for (int i = 0; i<10; i++)
        v.push_back(i);
    v.push_back(5);
    std::vector<int>::iterator it = findSecond(v.begin(), v.end(), 5);
    if (it != v.end())
        std::cout << *it << std::endl; // 5
}
```

Amennyiben `findSecond` nem talál két `v`-vel ekvivalens elemet, egy past-the-end iterátort ad vissza.

25.1.1. Megjegyzés. Lévén a konténert nem módosítjuk, bátran dolgozhatunk a nagyobb biztonságot nyújtó `const_iterator`-ral.

Azonban könnyű látni, hogy egyáltalán nem fontos információ az algoritmus szempontjából, hogy a `vector` `int`-eket tárol.

```
template<class T>
typename std::vector<T>::const_iterator
    findSecond(typename std::vector<T>::const_iterator begin,
               typename std::vector<T>::const_iterator end,
               const T &v)
{
    while(begin != end && *begin != v)
    {
        ++begin;
    }
    if (begin == end)
        return end;
    ++begin;
    while(begin != end && *begin != v)
    {
        ++begin;
    }
    return begin;
}
```

Sajnos sikerült elég kilométer függvény fejlécet sikerült írnunk, de a célt elértük.

Megállapítható, hogy itt még azt se kell tudnunk, minek az iterátorával dolgozunk, elegendő annyit tudnunk, hogy a `++` iterátorral lehet léptetni, és össze tudjuk hasonlítani őket az `!=` operátorral, azaz teljesítik egy input iterator feltételeit. Első körben vegyük az összehasonlítandó elem típusát külön template paraméterként.

```
template <class InputIt, class Val>
InputIt findSecond(InputIt begin, InputIt end, const Val &v)
{
    //...
}
```

Az STL algoritmusok iterátorokkal dolgoznak, azonban az, hogy minek az iterátorát használják, egyáltalán nem fontos tudniuk: elegendő annyi, hogy rendelkeznek azokkal az operátorokkal, melyeket fent felsoroltunk. Így például a mi listánk iterátorát ugyanúgy megadhatnánk a fenti függvények, mint egy `std::vector<int>`-ét.

A korábban elhangzottak alapján implementáljunk egy függvényt, mely egy iterátort léptet előre: Amennyiben a paraméterként kapott iterátor kategóriája bidirectional iterator, egyesével lépegessünk a kívánt helyre, random

access iterator esetén egyből ugorjunk oda. Ehhez használjuk ki azt, amit fentebb láttunk: az `std::iterator`-ból való öröklés hatására már le tudjuk kérdezni az iterátorunk kategóriáját!

```
template<typename BDIT>
BDIT algorithm(BDIT it, int pos, std::bidirectional_iterator_tag)
{
    for(int i = 0; i<pos; i++)
        ++it;
    std::cout << "Slow" << ' ';
    return it;
}

template<typename RAIT>
RAIT algorithm(RAIT it, int pos, std::random_access_iterator_tag)
{
    std::cout << "Fast" << ' ';
    return it + pos;
}

template<typename IT>
IT advance(IT it, int pos)
{
    typedef typename IT::iterator_category cat;
    return algorithm(it, pos, cat());
}

int main()
{
    std::vector<int> v;
    std::list<int> l;
    for (int i = 0; i<10; i++)
    {
        v.push_back(i);
        l.push_front(i);
    }
    std::cout << *advance(v.begin(), 3) << std::endl; // Fast 3
    std::cout << *advance(l.begin(), 3) << std::endl; // Slow 6
}
```

25.1.2. Megjegyzés. Az `std::vector` és `std::list`-nél felmerült iterátor lépegetésre találtunk megoldást: a fentihez hasonló működésű `std::advance` alkalmazható e célra.

Így tudjuk elérni azt, hogy különböző iterátor kategóriára különböző algoritmust használjunk (bár nyilván ez jóval hatékonyabban van megoldva a standard könyvtárban).

Ezek ismeretében vágjunk bele az STL algoritmusokba!

25.1.3. Megjegyzés. Cppreference-en megfigyelhető, hogy irtózatosan sok algoritmus van – csak úgy mint a konténereknél, a fontosabb algoritmusokról lesz szó, míg a többivel való ismerkedés gyakorlás végett az olvasóra bízom.

25.2. find

Az `std::find` segítségével az első adott értékű elemre hivatkozó iterátort kaphatunk. Keressük meg a 4-el ekvivalens elemet!

```
int main()
{
    std::set<int> s;
    for(int i = 0; i<10; i++)
        s.insert(i);
```

```
    std::set<int>::iterator result = std::find(s.begin(), s.end(), 4);
    if (result != s.end())
        std::cout << *result << std::endl; // 4
    else
        std::cout << "4 nem eleme" << std::endl;
}
```

25.2.1. Megjegyzés. Lévén az `std::set` egy bináris fa, így várhatóan a keresést logaritmikus idő alatt is meg tudjuk tenni: ezért szokás az ilyen speciális konténereknek egyedi find függvényt írni, ami az `std::set` esetében egy tagfüggvény.

```
std::set<int>::iterator it = s.find(42);
if (i != s.end())
    //...
```

Azonban megállapítandó, hogy míg az `std::find` az == operátorral végzi az összehasonlítást, addig az `std::set` a template paraméterként kapott rendezéssel! (ami alapértelemzetten a < operátortól függ.)

```
struct Circle
{
    int x, y, r;
    Circle(int _x, int _y, int _r) : x(_x), y(_y), r(_r) {}
};

bool operator<(const Circle &lhs, const Circle &rhs)
{
    return lhs.r < lhs.r;
}

bool operator==(const Circle &lhs, const Circle &rhs)
{
    return lhs.r == rhs.r && lhs.x == rhs.y && lhs.y == rhs.y;
}

int main()
{
    std::set<Circle> s;
    for(int i = 0; i<10; i++)
        s.insert(Circle(i, i + 1, i + 4));
    for(int i = 0; i<10; i++)
        s.insert(Circle(i, i + 1, i + 2));

    std::set<Circle>::iterator it1 = s.find(Circle(2, 3, 4));
    std::set<Circle>::iterator it2 = std::find(s.begin(), s.end(),
                                                Circle(2, 3, 4));

    std::cout << it1->x << ' ' << it1->y << ' ' << it1->r; // 0 1 4
    std::cout << it2->x << ' ' << it2->y << ' ' << it2->r; // 1 9 10
}
```

Ezzel semmi komolyabb gond nincs, de nem szabad meglepődni, amikor az ember ettől függően más megoldást kap mint amire számít.

25.3. sort és stable_sort

A következő példában próbáljuk rendezni egy `vector` elemeit!

```
std::vector<int> v {6,3,7,4,1,3};
std::set<int> s {6,3,7,4,1,3};
std::set<int> s1(v.begin(), v.end());
```

```
v.assign(s1.begin(), s1.end());
for(std::vector<int>::iterator it = v.begin(); it != v.end(); ++it)
{
    std::cout << *it << std::endl; // 1 3 4 6 7
}
```

25.3.1. Megjegyzés. Az fenti inicializálások a c++11-es újítás részei, sok magyarázatra gondolom nem szorulnak.

Most kihasználtuk, hogy az `std::set` alapértelmezetten rendezett: átpakoltuk az elemeket abba, majd visszaillesztettük az eredeti konténerbe. No persze, ez minden, csak nem hatékony. Ráadásul az egyik 3-as kiesett: az `std::set` megszűrte az elemet, mert ekvivalenseket nem tárol.

Az ilyen jellegű barkácsolások sose fogadhatóak el hatékonyság szempontjából. Ismerkedjünk meg az ennél sokkal hatékonyabb `std::sort`-al!

```
std::vector<int> v {6,3,7,4,1,3};
std::sort(v.begin(), v.end());
for(std::vector<int>::iterator it = v.begin(); it != v.end(); ++it)
    std::cout << *it << ' '; // 1 3 3 4 6 7
```

Az `std::sort`, ha külön paramétert nem adunk neki, az < operátor szerint rendez.

Rendezzük a fenti elemeket úgy, hogy a > operátor szerint legyenek sorban!

```
struct Greater
{
    bool operator()(int lhs, int rhs) const
    {
        return lhs > rhs;
    }
};

int main()
{
    std::vector<int> v {6,3,7,4,1,3};
    std::sort(v.begin(), v.end(), Greater());
    for(std::vector<int>::iterator it = v.begin(); it != v.end(); ++it)
        std::cout << *it << ' '; // 7 6 4 3 3 1
}
```

25.3.2. Megjegyzés. Ez jól demonstrálja, hogy az `std::sort`, csak úgy mint a legtöbb STL algoritmus, számos overload-al rendelkezik.

Az `std::sort` az ekvivalens elemek sorrendjét nem (feltétlenül) tartja meg: rendezés után azoknak a sorrendje nem definiált.

```
struct StringLength
{
    bool operator()(const std::string &lhs, const std::string &rhs) const
    {
        return lhs.size() < rhs.size();
    }
};

int main()
{
    std::vector<std::string> v;
    v.push_back("ADA");
    v.push_back("Java");
    v.push_back("1234567");
    v.push_back("Maci");
```

```
    v.push_back("C++");
    v.push_back("Haskell");

    std::sort(v.begin(), v.end(), StringLength());

    for(std::vector<std::string>::iterator it = v.begin(); it != v.end(); ++
        it)
         std::cout << *it << ' ';
}
```

Lehetséges output: `C++, ADA, Java, Maci, Haskell, 1234567.`
Amennyiben fontos, hogy az ekvivalens elemek relatív sorrendje megmaradjon, használjunk `std::sort` helyett `std::stable_sort`-ot, mely pontosan ezt csinálja.

25.3.3. Megjegyzés. Ennek nyilván ára is, van, általában az `std::stable_sort` kevésbé hatékony.

Link: `http://en.cppreference.com/w/cpp/algorithm/sort`.

25.4. remove

Az `std::remove` algoritmus a konténerben lévő elemek törlését segíti elő. Töröljük ki egy vectorból az összes 3-al ekvivalens elemet!

```
std::vector<int> v{1,2,3,3,4,5,6};
std::cout << v.size() << std::endl; // 7
std::remove(v.begin(), v.end(), 3);
std::cout << v.size() << std::endl; // 7
```

Legnagyobb meglepetésünkre, ez semmit se fog törölni: az `std::remove` átrendezi a konténert, úgy hogy a konténer elején legyenek a nem törlendő elemeket, és utána a törlendőek, és az első törlendő elemre visszaad egy iterátort. Ennek segítségével tudjuk, hogy ettől az iterátortól a past-the-end iteratorig minden törlendő.

```
std::vector<int> v{1,2,3,3,4,5,6};
std::cout << v.size() << std::endl; // 7
auto it = std::remove(v.begin(), v.end(), 3);
v.erase(it, v.end());
std::cout << v.size() << std::endl; // 5
```

25.4.1. Megjegyzés. Ebben a kódban van pár c++11-es újítás is: az `auto` kulcsszó megadható konkrét típus helyett. Ilyenkor az egyenlőségjel bal oldalán lévő objektum típusára helyettesítődik a kulcsszó. Példaképp, fent a fordító meg tudja határozni, hogy az `std::remove`-nak a visszatérési értéke itt `std::vector<int>::iterator` lesz, így a kód azzal ekvivalens, mintha ezt írtuk volna:

```
std::vector<int>::iterator it = std::remove(v.begin(), v.end(), 3);
```

Bár az `auto` kulcsszót sokáig lehetne még boncolgatni, legyen annyi elég egyenlőre, hogy a template paraméter dedukciós szabályok szerint működik (c++11et nem szükséges tudni a vizsgához).

25.5. Végszó az algoritmusokhoz

Megjegyzendő, hogy nem minden algoritmus működik minden iterátor típussal (pl. az `std::sort` random access iteratort vár), és vannak egyes algoritmusok, melyeknek speciális előfeltételei is vanak (például az `std::unique` egy rendezett konténert vár).

Fontos az is, hogy ezek az algoritmusok általában nagyon hatékonyak, így gyakran nem is érdemes saját magunktól megírni.

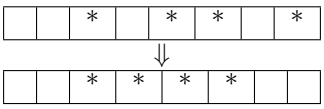
Gyakran nincs szükség arra, hogy a függvénynevek elé kiírjuk, melyik névtérből származnak.

```
//...
auto it = remove(v.begin(), v.end(), 3);
//...
```

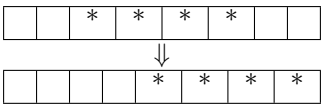
Itt a fordító a függvény paraméterekből ki tudja találni , hogy milyen névtérből van az algoritmus. Ezt *Argument Dependent Lookup*-nak nevezzük, vagy röviden ADL-nek. Mivel a `vector` ugyanúgy az `std` névtérből származik, és azt adtunk meg paraméternek, tudni fogja a fordító, hogy a `remove`-ot is abban a névtérben kell keresni.

25.6. Gyakorló feladat

Vegyünk egy konténert, és jelöljünk meg benne bár elemet, melyektől azt szeretnénk, hogy legyenek egymás mellett, de minden más elem sorrendje ne változzon.



Próbáljuk a csillaggal jelölt elemet egymás mellé tenni: ezt megtehetjük pl. két `stable_partition`-nel, mert az ekvivalens elemek sorrendjét nem változtatják.
Most csináljuk azt, hogy a kijelölt elemek a konténer végén legyenek, de a többi elem sorrendje ne változzon! Itt használhatjuk az előbbi algoritmusok után az `std::rotate`-et.



„Általában az emberek nem szeretik, hogyha kicsesznek velük, és ha kicseszel a kollégáiddal, morcosak lesznek”
/Horváth Gábor/

26. Objektum orientált programozás

Manapság az egyik legnépszerűbb programozási paradigma a objektum orientált programozás (*object oriented programming*, röviden *OOP*). Bár sokféleképpen definiálhatjuk, e három dolgot azonban általában megköveteljük egy objektum orientált kódtól:

- enkapszuláció, azaz az adatok egybefoglalása
- kód újrafelhasználás
- adat elrejtés

Fontos, hogy az objektum orientáltságot ne kössük feltétlenül az osztályokhoz, ezeket a követelményeket ugyanis ha körülményesen is, de nélkülük is meg tudjuk oldani. Példaképp, C-ben létrehozhatunk egy egyszerű rekordot, melynek deklarációját és a hozzá tartozó műveleteket eltároljuk egy header fájlban, magát a rekordot és a függvényeket csak egy fordítási egységben definiáljuk. Ezzel elértük az enkapszulációt, az adatok rejtve vannak, és egy ilyen rekordot el tudunk tárolni egy másik rekordban is könnyedén, ezzel megvalósítva a kód újrafelhasználást.
Így bátran kijelenthetjük, hogy a C nyelv egy olyan nyelv, mely támogatja az objektum orientált programozást. Azonban az is megállapítható, hogy ez nagyon nehézkes, és a legtöbb nyelv, melyre objektum orientáltként hivatkozunk (Pl. Java) sokkal erősebb eszközökkel rendelkeznek ennek támogatására.
Az objektum orientált C++ kódolásról nagyon sokat lehet mesélni, az elkövetkező néhány szekcióban áttekintjük valamennyi fontosabb részét, és hogy a fenti elveket hogyan tudjuk megvalósítani vele.

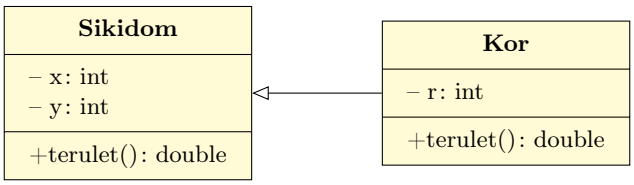
26.1. Öröklődés (*inheritance*)

Próbáljunk megalkotni egy osztályt, mellyel egy általános síkidomot tudunk jellemezni!

```
class Sikidom
{
    int x, y; // a síkidom koordinátái
public:
    double terulet() const
    {
        // ???
    }
};
```

Az egyik legelső dolgunk legyen az, hogy írunk egy területszámító függvényt. Azonban egy általános síkidomnak nem tudjuk megadni csak így a területét. Esetleg ennek az típusnak létrehozhatnánk egy *altípusát*, mondjuk egy kört, amely rendelkezik a síkidom minden tulajdonságával, és még a körre vonatkozó adatokkal. Ehhez **öröklődést** fogunk alkalmazni:

```
class Kor : public Sikidom
{
    double r;
public:
    double terulet() const
    {
        return r * r * 3.14;
    }
};
```



A `Sikidom` és a `Kor` osztály kapcsolata, melyet a fent látható nyíllal fejezünk ki: a nyíl a leszármazott osztályból az ősosztályba mutat.

Egy adott `struct`-ból illetve `class`-ból a fent látható módon tudunk örökölni: az új osztály neve után kettőspontot írunk, opcionálisan megadjuk a hozzáférés típusát, mely ebben az esetben public (erről nemsokára bővebben lesz szó), és az osztály nevét, melyből örökölni szeretnénk.
Az öröklődés hatására a `Kor` osztály `Sikidom` összes adattagját és metódusát megörökli. Ilyenkor azt mondjuk, hogy a `Kor` osztály `Sikidom` **leszármazottja** (*derived class*), és a `Sikidom` osztály `Kor` **őse** (*base class*). Példaképp, így tudunk hivatkozni `Kor`-ön belül `Sikidom` egy adattagjára:

```
class Kor : public Sikidom
{
    double r;
public:
    // ...
    void f()
    {
        std::cout << Sikidom::x << std::endl;
        std::cout << x << std::endl;
    }
};
```

A fenti kóddal kapcsolatban két dolog állapítható meg: Mivel `x` nevű objektum csak egy darab van az osztályban, ezért nem kell explicit módon `Sikidom`-on keresztül hivatkoznunk `x`-re. A másik dolog, hogy ez a kódrészlet sajnos nem helyes, fordítási hibát okoz: `x` ugyanis privát adattagja `Sikidom`-nak, és privát adattaghoz csak és kizárólag az adott osztály fér hozzá, még a leszármazott sem.
Ez probléma hisz mi `Kor`-t azért hoztuk létre, hogy speciálisabb feladatokat tudjuk végrehajtani `Sikidom` tulajdonságaival. Írjuk át ezeknek az adattagoknak a védettségét protectedre:

```
class Sikidom
{
protected:
    int x;
    int y;
public:
    double terulet() const {}
};
```

A `protected` tagokhoz csak az adott osztály fér hozzá, és azon osztályok, melyek ebből örökölnek.

26.2. Publikus öröklés

Öröklődésnél 3 különöző hozzáférési típust különítünk meg: `public, protected, private`. Ezeket a kulcsszavak az öröklődés kontextusában *access specifier*-nek nevezzük, és mind arra van kihatással, hogy az ősosztályból örökölt adattagok és metódusok milyen láthatóságot kapnak a leszármazottban, a következőféleképpen:

Öröklődés típusa:	public	protected	private
`public` örökölt adatok/metódusok	public	protected	private
`protected` örökölt adatok/metódusok	protected	protected	private
`private` örökölt adatok/metódusok	private	private	private

Az alábbi táblázat mutatja, hogy öröklődés után a bázisosztályban lévő adattagok/metódusok láthatóak-e a leszármazott osztályban:

Öröklődés típusa:	public	protected	private
Saját adatok/metódusok	igen	igen	igen
Örökölt adatok/metódusok (public)	igen	igen	igen
Örökölt adatok/metódusok (protected)	igen	igen	igen
Örökölt adatok/metódusok (private)	nem	nem	nem

A következőkben tekintsük a publikus öröklés tulajdonságait.

Az, hogy publikusan örököltünk, azt is jelenti, hogy egy altípus-t (*subtype*) hoztunk létre, és mint egy altípus, minden helyre, ahol `Sikidom`-ot szeretnénk használni, `Kor`-t is használhatjuk.

26.2.1. Megjegyzés. Az öröklődés típusa alapértelmezetten (azaz ha nem írjuk ki az access specifiert) ha `class`ból öröklünk `private`, ha `struct`ból akkor `public`.

26.2.2. Megjegyzés. A publikus öröklődést szokás **az egy** (*is a*), privátot **van egy** (*has a*) kapcsolatnak is hívni.

```
Kor k;
Sikidom s;
s = k; //ok
```

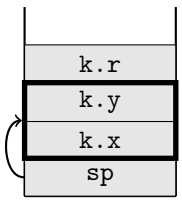
Azt már letisztáztuk, hogy `Kor` `Sikidom` altípusa, de azonban ez akkor is két különbötő típus, hogyan lehet, hogy nem kapunk fordítási idejű hibát?
A válasz az, hogy a fordító meg tudja oldani e két típus közötti értékadást. Mivel `Kor`-ben benne van maga `Sikidom` is, ezért meghívja az ahhoz tartozó értékadó operátort, és a `Kor` osztály `Sikidom`-ból örökölt adattagjait adja majd értékül, ebben az esetben (lévén nincs speciális értékadó operátor megírva `Sikidom`-ban) ez fog történni:

```
s.x = k.x;
s.y = k.y;
```

Ez azonban magával hordozza azt is, hogy minden információ, ami `Kor`-re specifikus (pl. sugár (`r`)) elveszik. Ez a folyamatot *slicing*.
A slicing láthatóan olyan jelenség, mely az esetek többségében elkerülendő. Szerencsére van lehetőség arra, hogy egy `Kor` típusú objektumra egy `Sikidom` típusú objektumon keresztül hivatkozzunk:

```
Kor k;
Sikidom *sp = &k;
Sikidom &sr = k; //a referencia ugyanúgy működik, mint a pointer
```

A fenti példában `sp` *statikus típusa* `Sikidom*`, a dinamikus típusa pedig `Kor`. Ez azt jelenti, hogy a fordító biztosan tudja már fordítási időben hogy `sp`-nek milyen típusa lesz, azonban azt nem tudhatja, hogy éppen milyen típusú objektumra mutat a `Sikidom`-hoz tartozó öröklődési fában, ez csak futási időben derülhet ki.



`sp` pointer csak a `k` objektum `Sikidom` részére tud hivatkozni.

Ez meg is felel az eredeti célunknak – amennyiben csak egy `Sikidom` objektummal szeretnénk dolgozni, addig mindegy is, hogy van-e a azon túl extra adat. Ha például egy olyan barát függvényt írunk, mely `Sikidom`-ok koordinátáit íratja ki, fölösleges információ ahhoz a sugár, vagy bármilyen egyéb tulajdonság.
Mi fog azonban történni, ha meghívjuk a `terulet` metódust?

```
Kor k;
Sikidom *sp = &k;
sp->terulet();
```

Ilyenkor a `Sikidom`-hoz tartozó `terulet` függvény kerül meghívásra. Ennek az oka az, hogy C++ban alapértelmezetten mindig a pointer/referencia statikus típusa szerint kerülnek a tagfüggvények meghívásra. Ennek hatékonyságai okai vannak, ugyanis így azt, hogy melyik függvényt kell meghívni (`Sikidom::terulet` vagy `Kor::terulet`) már fordítási időben meg lehet határozni. Ahhoz, hogy kikényszerítsük, hogy a fordító a dinamikus típusnak megfelelő függvényt hívja meg, a `virtual` kulcsszóra lesz szükségünk.

26.3. Virtuális függvények

Azokat az osztályokat, melyek legalább egy virtuális függvénnyel rendelkeznek, polimorfikus osztálynak (*polymorphic class*) nevezzük.

```
class Sikidom
{
    //...
public:
    virtual double terulet() const //virtuális!
    {
        //...
    }
};

class Kor : public Sikidom
{
    double r;
public:
    double terulet() const
    {
        return r * r * 3.14;
    }
};
```

Minden metódus, mely öröklődésnél a bázisosztály egy virtuális metódusával megegyező névvel, visszatérési értékkel, paraméterlistával és konstansággal rendelkezik, implicit módon virtuális lesz. Konstruktor sose lehet virtuális. (pl.: `Kor`-ben `terulet` implicit módon virtuális lesz, mert `Sikidom`-ban is az volt).

A virtualitás következtében ha egy `Sikdiom` statikus típusú pointeren vagy referencián szeretnénk meghívni a `terulet` függvényt, akkora fordító futási időben leellenőrzi, hogy mi a mutatott terület típusa (dinamikus típusa) és az annak megfelelő függvényt hívja meg. Ez azt is jelenti, hogy ha egy öröklődési fában egy adott gyökérnek van egy virtuális függvénye, akkor minden belőle származó gyereknek az az adott virtuális függvénye leárnyékolja azt az adott függvényt.
A bázisosztály függvényének leárnyékolása azonban nem azt jelenti, hogy azt a függvényt nem lehet meghívni. Ha explicit módon jelezzük a fordítónak, hogy a bázisosztályban definiált metódust szeretnénk meghívni, akkor minden virtualitás ellenére is az lesz meghívva:

```
int main()
{
    Kor k;
    std::cout << k.Sikidom::terulet();
}
```

Ezt explicit névfeloldásnak (*explicit scope resolution*) nevezzük.

26.4. Tisztán virtuális függvények

Matematikailag, ha egy síkidomot háromszögekre felvágunk, meg tudjuk mondani a területét, ez azonban a jelenlegi példánkban ennyi adat segítségével aligha lehetséges. Ezért gyakran úgy dönthetünk, hogy egy adott függvénynek csak a fejlécét (*interface*) örököltetjük, de implementációját nem.

A fenti osztálynál azonban probléma, hogyha valaki létrehoz egy `Sikidom` típusú objektumot, és meghívja a `terulet` függvényt, jogosan várja, hogy az ki is számítsa neki. Ennek megkerülésére elérhetjük, hogy fordítási hibát kapjon, aki ilyennel próbálkozna: `Sikidom`ban `terulet`-et tisztán virtuálissá tesszük:

```
class Sikidom
{
    //...
public:
    virtual double terulet() const = 0; //tisztán virtuális!
};
```

Mostmár `terulet` tisztán virtuális (*pure virtual*) `Sikidom`-ban. Az olyan osztályokat, melyekből nem lehet objektumot létrehozni, absztraktnak nevezzük. Az olyan osztályokból, melyek tartalmaznak tisztán virtuális metódusokat, nem tudunk objektumot létrehozni.

26.4.1. Megjegyzés. Egy osztály úgy is lehet absztrakt, hogy `protected`-dé tesszük a konstruktorait (erről majd később). Az egyszerűség kedvéért, az elkövetkezendő szekciókban ha absztrakt osztályról beszélünk, gondoljunk egy tisztán virtuális metódussal rendelkező osztályra.

Egy olyan osztály, mely egy absztrakt osztályból örököl, ahhoz, hogy példányosítható legyen (lehessen olyan típusú objektumokat létrehozni), felül kell írnia az összes abban található tisztán virtuális függvényt.

```
int main()
{
    Sikidom s; //nem ok, fordítási idejű hiba: Sikidom absztrakt
    Kor k; //ok
    Sikidom *sp = &k;
    Sikidom &sr = k;

    sp->terulet(); //ok, Kor::terulet()-et hívja meg
    sr.terulet(); //szintén
}
```

Mint láthatjuk, absztrakt típusú pointert és referenciát lehet létrehozni, így a polimorfizmus összes előnyével élhetünk, továbbá mindenkit rá tudunk kényszeríteni arra, hogy a `terulet` függvényt implementálja, ha `Sikidom`-ból örököl és példányosítani szeretné azt az osztályt.
Persze, ettől mi még írhatunk definíciót egy tisztán virtuális függvényhez:

```
class Sikidom
{
    //...
public:
    virtual double terulet() const = 0;
};

double Sikidom::terulet()
{
    return -1; //negatív számmal jelezzük a számolás sikertelenségét
}
```

Figyeljük meg, hogy egy tisztán virtuális függvényt csak a deklarációtól külön tudunk definiálni.
Ez a trükk például akkor hasznosítható, ha egy van egy absztrakt bázisosztályunk, melynek szeretnénk, hogy öröklés után egy függvényét mindenki definiáljon felül, de akarunk adni hozzá egy alap implementációt.

```
class Sokszog : public Sikidom
{
    public:
    double terulet() const
    {
        return Sikidom::terulet();
    }
};
```

26.4.2. Megjegyzés. Nyilván ebben az esetben nem célszerű definiálnunk `Sikidom::terulet`-et, hisz ezzel csak lehetőséget adunk arra magunknak és mindenkinek, aki a `Sikidom` osztályt használja, hogy lábon lője magát.

A későbbiekben tekintsünk `Sikidom`-ra úgy, mintha nem lenne tisztán virtuális függvénye.

26.5. Konstruktorok öröklődésnél

Öröklődésnél a konstruktorok és destruktorok működése is újabb kihívásokat jelent. Lévén értelmetlen úgy létrehozni egy síkidomot, hogy nem adunk meg neki pozíciót (azaz, nem inicializáljuk az `x` és `y` változót), hozzunk létre e célra egy konstruktort. Így azt is el tudjuk érni, hogy a fordító nem generáljon default konstruktort:

```
class Sikidom
{
protected:
    int x, y;
public:
    Sikidom(int x, int y) : x(x), y(y) {}
    virtual double terulet() const {}
};

class Kor : public Sikidom
{
    double r;
public:
    double terulet() const {/* ... */}
};

int main()
{
    Sikidom s(1,2); //ok
    Kor k; //fordítási hiba
    Kor k2(1,2); //fordítási hiba
}
```

Mivel konstruktort nem írtunk `Kor`-nek, így a fordító generál nekünk egyet, melyben megpróbálja `Sikidom`-hoz tartozó adattagokat `Sikidom` paraméter nélküli konstruktorával meghívni. Azonban mivel nincs neki ilyen, fordítási idejű hibát kapunk. Ilyenkor csak egy lehetőségünk van a `Sikidom` osztály módosítása nélkül, írnunk kell `Kor`-nek egy konstruktort, melynek inicializációs listájában meghívjuk `Sikidom` egyetlen, két `int`-et váró konstruktorát.

```
class Sikidom
{
protected:
    int x, y;
public:
    Sikidom(int x, int y) : x(x), y(y) {}
    virtual double terulet() const {}
};

class Kor : public Sikidom
{
    double r;
public:
    Kor(int x, int y, int r) : Sikidom(x,y), r(r) {}
    double terulet() const {/* ... */}
};

int main()
{
    Sikidom s(1,2); //ok
```

```
    Kor k2(1,2); //ok
}
```

Mi történni a copy konstruktorokkal? Hasonló helyzetben, ha egy kört másolunk, a síkidom default copy konstrukora fog meghívódni, kivéve, ha az felül van definiálva. Azaz alapértelmezetten rendre másolásnál először meghívódik a `Sikidom` copy konstruktora, melyet a `Kor`-é követ.

```
class A {};
class B : public A {};
class C : public B {};
class D : public C {};
```

Konstruktorok hívási sorrendje: `A -> B -> C -> D`

Előfordulhat olyan helyzet, mikor nem akarjuk, hogy az adott osztályt példányosítsák. Például a fent erre egy módszer volt a tisztán virtuális metódusok használata. Egy másik módszer az, ha az összes konstruktort védetté tesszük.

```
class Base
{
protected:
    Base() {}
};

class Derived : public Base {};
```

Ebben az esetben `Derived` meg tudja hívni `Base` konstruktorát, így példányosítható lesz, de önmagában `Base` nem.

26.6. Destruktorok öröklődésnél

Ha a megfelelő `terulet` függvény csak akkor hívódott meg, ha a `terulet` függvényt `Sikidom`-ban virtuálissá tettük, hogyan fognak viselkedni a destruktorok? A válasz az, hogy azoknak a működése is közel azonos e téren, azaz nem a megfelelő destruktor fog lefutni, egy bázisosztályon keresztül próbálunk egy leszármazottat megsemmisíteni.

```
Sikidom *s = new Kor(1,2);
delete s; //undefined behaviour
```

A nem definiált viselkedés ellenére azért lehet nagyjából tudni, mi fog történni: mivel `s` statikus típusa `Sikidom`, az objektumon annak a destruktora fog lefutni. Ez az esetek többségében azt jelenti, hogy minden extra adat ezen kívül elszivárog.
Javítsuk is ezt gyorsan:

```
class Sikidom
{
protected:
    int x;
    int y;
public:
    Sikidom(int x, int y) : x(x), y(y) {}
    virtual ~Sikidom() {}
    virtual double terulet() const;
};
```

Bár örökölni közel minden osztályból lehet (ezalól kivétel lehet egy osztályon belüli privátként deklarált inline class-ok), ettől még nem mindig bölcs döntés az. Példaképp örökölni STL konténerekből nagy galibát okozhat, hisz azoknak a destruktora nem virtuális.

Ha biztosra akarunk menni, deklaráljunk minden destruktort virtuálisnak. Azonban tartsuk észben, hogy csakúgy mint minden virtuális függvény, ez futási idejű költséggel párosul. Ha 100%, hogy az osztályunkból nem fognak örökölni, bátran hagyjuk annak destruktorát `virtual` kulcsszó nélkül.

A fenti `A, B, C, D` osztályokat tekintvén, a destruktorok lefutási sorrendje ellentétes: `D -> C -> B -> A`.

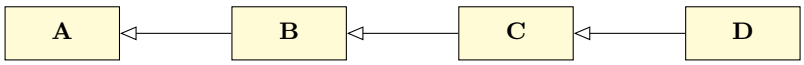
26.7. Bővebben a slicing veszélyeiről

A slicing-ról már volt szó, azonban veszélyesebb az talán, mint elsőre gondolnánk. Emlékeztetőként, akkor fordul ez elő, ha egy objektumot egy, a bázisosztály objektumának adjuk **értékül**. Ilyenkor minden speciális adat elveszik, nem fogunk tudni hivatkozni specifikus metódusokra, de ami még veszélyesebb, hogy a virtuális függvényeket ugyanúgy meg fogjuk tudni hívni, azonban azok mást fognak csinálni, mint amire számítunk.

```
struct Base
{
    virtual void print() const
    {
        std::cout << "Base!" << std::endl;
    }
};

struct Derived : public Base
{
    void print() const
    {
        std::cout << "Derived!" << std::endl;
    }
};

void printRef(const Base& b)
{
    b.print();
}

void printVal(const Base b)
{
    b.print();
}

int main()
{
    Base b;
    Derived d;
    printRef(b); //Base!
    printRef(d); //Derived!
    printVal(b); //Base!
    printVal(d); //Base!
}
```

Láthatjuk, ebbe a problémába belefutni könnyebb mint gondolnánk. Ez is plusz egy indok, miért preferáljuk a referencia szerinti átvételt, és érték szerint csak akkor veszünk át, ha nagyon indokolt. E téren a kivételezésnél is jobb ha figyelünk!

```
int main()
{
    try
    {
        throw Derived();
    }
    catch(const Base& b) //referencia!
    {
        b.print(); //Derived!
    }
}
```



Függvény...	Eredeti típus	Módosított típus		
...paraméterei	A*	B*	C*	D*
	B*	A*	C*	D*
	C*	A*	B*	D*
	D*	A*	B*	C*
...visszatérési érte	A*	B*	C*	D*
	B*	A*	C*	D*
	C*	A*	B*	D*
	D*	A*	B*	C*

Az alábbi táblázat összefoglalja az itt megállapítottakat.
A fenti osztályhierarchiában `A` a legáltalánosabb, `D` a legspeciálisabb.
(piros: a módosítás nem megfelelő, zöld: a módosítás jó lesz):

26.8. Kód módosítása polimorfizmusnál

A polimorfikus osztályok használatának egyik nagy előnye, hogy már megírt függvényeket módosíthatunk anélkül, fordítása hiba keletkezzék bárhol, ahol az adott függvény már alkalmazásban van.

```
void f(Kor* k)
{
    //...
}
```

Lehet, az implementáció egy pontján úgy döntünk, hogy az `f` függvény várjon inkább `Sikdiom`-okat, mert nem használ fel semmilyen `Kor`-re specifikus adatot.

```
void f(Sikidom* k)
{
    //...
}
```

Ez azért nem okozhat problémát, mert minden olyan objektum, melynek típusa `Kor` vagy `Kor`-ből örökölt, ugyanúgy átkonvertálható `Sikidom*` típusra, mint `Kor*` típusra.
Ennek a tanulsága az, hogy függvény paramétereknél **felfelé** az öröklődési láncon lehet haladni. Értelemszerű okokból `Sikidom*`-ról nem válthatunk `Kor*`-ra a fordítási hiba veszélye nélkül, hisz egy `Sikidom` típusú objektum nem adható át egy olyan függvénynek.

Tekintsük most a függvények visszatérési értékét:

```
Sikidom* makeObject() {}
```

Hasonlóan, döntsünk úgy hogy az `makeObject` függvény `Kor*`-t adjon vissza.

```
Kor* makeObject() {}
```

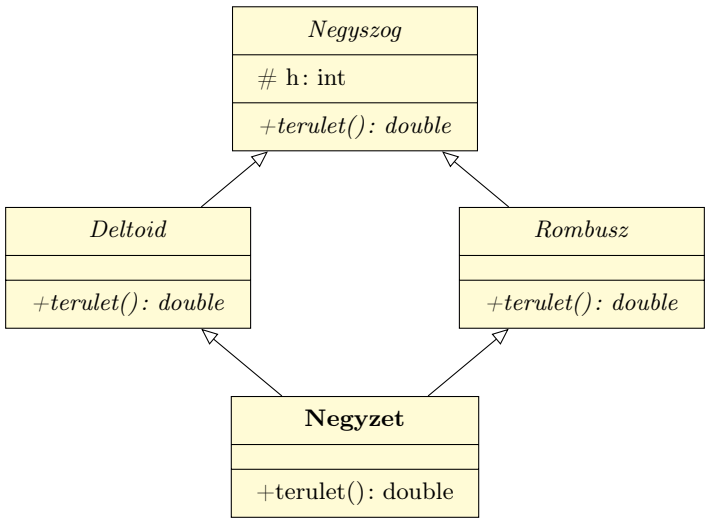
A fentiekhez hasonló okokból, ez sem okoz majd problémát. Megállapítható, hogy egy függvény visszatérési értéke egy adott öröklődési fán **lefelé** bátran módosulhat.

26.8.1. Megjegyzés. Ez mutatja jól a publikus öröklődésnek az *az egy* relációját, hisz minden kör egy síkidom, de nem minden síkidom kör.

26.9. Többszörös öröklődés

Viszonylag kevés nyelv engedi meg, hogy egyszerre több osztályból is örököljünk. A C++ azonban nem ilyen:

```
class BaseOne {};
class BaseTwo {};

class Derived : public BaseOne, public BaseTwo {};
```



Egy példa a gyémántöröklésre.

A többszörös öröklés egy nagyon erős eszköze a nyelvnek, hisz az *az egy* relációt több másik osztállyal is fel tudjuk állítani. Példaképp, ha adott egy `Bird` és egy `NonFlyingAnimal` osztály, a `Penguin` örökölhetne mindkettőből, hisz minden pingvin *az egy* madár, és minden pingvin *az egy* röpképtelen állat.
Azonban általában igaz az, hogy a többszörös öröklődésnél több a galiba mint a haszon, így hacsak nincs nagyon jó okunk rá, ne erőltessük. Nehezen lesz velük átlátható, hogy milyen lesz egy pointer dinamikus típusa, rászorulhatunk side cast-okra (erről később), és egyéb nem elegáns módszerekre.

A többszörös öröklés igazi fő gonosza talán a gyémántöröklődés, melynek használata közel minden esetben elkerülendő.

```
class Negyszog
{
protected:
    double h; //height
public:
    virtual double terulet() = 0;
};

class Rombusz : public Negyszog
{
public:
    //az egyszerűség kedvéért maradjon tisztán virtuális
    double terulet() = 0;
};

class Deltoid : public Negyszog
{
public:
    double terulet() = 0;
};

class Negyzet : public Deltoid, public Rombusz
{
public:
    double terulet()
    {
        return h * h;
    }
};
```

A legutóbbi osztály természetesnek tűnhet, hisz egy négyzet deltoid is, meg rombusz is, így szeretnénk, ha

minden függvény, ami `Deltoid`-ot vagy `Rombusz`-t vár, `Negyzet`-et is elfogadjon. Azonban a fenti kód nem fordul le, hisz mind `Rombusz`-ban, mind `Deltoid`-ban van egy teljes `Negyszog` is. Így `Negyzet` kétszer fogja tartalmazni annak összes adattagját, így `h`-t is, és a fordító nem tudja kitalálni, mi épp melyikre gondolunk. 2 kézenfekvő megoldás lehet:

```
class Negyzet : public Deltoid, public Rombusz
{
public:
    double terulet()
    {
        return Deltoid::x * Deltoid::x;
    }
};
```

Azonban leggyakrabban nem cél, hogy egy leszármazott bármelyik őst kétszer tartalmazza. Erre megoldás lehet a virtuális öröklődés.

```
class Negyszog
{
protected:
    double x;
public:
    virtual double terulet() = 0;
};

class Rombusz : virtual public Negyszog //virtuális öröklés!
{
public:
    double terulet() = 0;
};

class Deltoid : virtual public Negyszog
{
public:
    double terulet() = 0;
};

class Negyzet : public Deltoid, public Rombusz
{
public:
    double terulet()
    {
        return x * x;
    }
};
```

Ezzel garantáltuk is ezt, de milyen áron? A virtuális öröklődésnek, csak úgy mint a virtuális metódusoknak futási idejű költsége van, így ha egy mód van rá, kerüljük. Az öröklődési gráfot is nehéz átlátni és értelmezni, ha tartalmaz gyémánt öröklést.
Annak ellenére, hogy a gyémánt öröklődés erősen ellenjavallott dolog, van kivétel még a standard könyvtáron belül is: az `iostream`. A beolvasó és kiíratós stream-ek közös tulajdonságait tartalmazza `iosbase`, melyből örököl `istream` és `ostream` is, és mindkettőből az `iostream`.

26.10. Privát és védett öröklődés

Lévén az esetek **erősen** túlnyomó többségében publikusan öröklünk, és igény is nagyon kevés van ettől eltérni, így ez a szekció nem részleteiben fogja tárgyalni e két öröklődési módot.

```
struct Base
{
    int x;
```

```
    virtual ~Base() {}
};
struct Derived : private Base {}; //private inheritance
struct DerivedTwo : public Derived {}; //public inheritance

int main()
{
    Derived *d = new DerivedTwo(); //ok
    Base *b = d; //fordítási hiba
}
```

A fenti kódrészlet azzal a hibaüzenettel nem fog lefordulni, hogy a `Base` egy nem elérhető bázisa `Derived`-nak. Ennek oda egyszerű: mivel privátan örököltünk, ezért `x` nem hozzáférhető `DerivedTwo`-ban, azonban hozzá tudnánk férni, ha `Base`-ként tudnánk rá hivatkozni.
Ezt értelmezhetjük úgy is, hogy a a privát/védett öröklés „elvágja” az öröklődési fát olyan értelemben, hogy a „vágáson” keresztül nem lehet leszármazottakat ősosztály típusba konvertálni.

Privát örökléssel szokás például tartalmazást is kifejezni. Bár leggyakrabban ha egy másik osztály adattaggá teszünk, kifizetődőbb, van eset, amikor pl. az interface egy részét elérhetővé akarjuk tenni.
Egy másik példa lehet, amikor egy olyan osztályból öröklünk, melynek nincs virtuális destruktora. Ezzel elérhetjük, hogy véletlenül se tudjunk átkonvertálni az ősosztály típusára, és ezzel nem elkerülhetjük a nem definiált viselkedést.

```
class MyVector : private std::vector<int>
{
public:
    using std::vector<int>::push_back; //C++11
};
```

A védett öröklés szinte univerzálisan sosem használt.

26.11. C++11

C++11-ben lehetőségünk van picit kényelmesebbé tenni az életünket, például az `override` kulcsszóval, mely garantálja, hogy az adott függvény felülír egy másikat.

```
class Base
{
public:
    virtual void f();
};

class Derived : public Base
{
public:
    virtual void f() override {}
};
```

Amennyiben a felülírás mégse sikerült (`f` nem virtuális, nem egyezik a paraméterlista/konstansság), fordítási idejű hibát kapunk, mely mindig sokkal jobb, mint a futási idejű.

Hasonlóan hasonló kulcsszó a `final`, mely megtiltja az öröklődést, így bátran használhatunk nem virtuális destruktorokat.

```
class Base final
{
public:
    virtual void f();
};

class Derived : public Base {}; //fordítási idejű hiba
```

Amennyiben örököltünk egy osztályból, mely privát adattagokat tartalmaz, azokhoz nem fogunk tudni hozzáférni. Ilyenkor rá vagyunk kényszerülve get/set függvények alkalmazására, ha azok meg vannak írva (szempont lehet megírásuk szempontjából ez is).

27. Cast-ok

Implicit konverziókkal számtalanszor találkozunk, például ha egy lebegőpontos számot egy egész számnak adunk értékül, vagy fordítva. Konverzió volt az is, amikor egy konverziós operátor vagy konverziós konstruktor segítségével egy osztályhoz tartozó objektumot egy másik osztály típusúnak adtunk értékül. A pilomorfikus osztályoknál meg számtalanszor éltünk már vele.

27.1. dynamic_cast

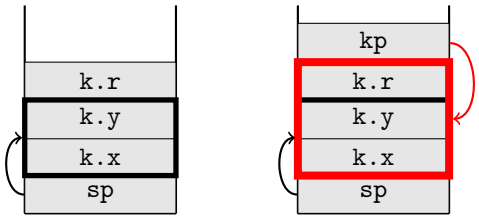
Néha abba a kellemetlen helyzetbe kerülhetünk, hogy van egy adott pointerünk/referenciák, és a dinamikus típusa `Kor`, azonban statikus típusa `Sikidom`, de mégis fontos, hogy `Kor`-re vonatkozó dolgokat használjunk. Ilyenkor segíthet a `dynamic_cast` mely képes ezt a konverziót végrehajtani:

```
void f(Sikidom *sp)
{
    Kor *kp = dynamic_cast<Kor*>(sp);
}
```

Megfigyelhető, hogy kacsacsőrbe írtuk azt a típust, amelyre konvertálunk, és utána gömbölyű zárójelbe azt, melyből konvertálnánk. A `dynamic_cast`-ot **csak és kizárólag** polimorfikus osztályoknál alkalmazhatjuk.

27.1.1. Megjegyzés. A `dynamic_cast` előtt nincs `std`. Ennek oka, az, hogy a `dynamic_cast` egy operátor, és nem egy standard függvény.

Fontos megállapítás, hogy ez a cast egy adott pointer/referencia dinamikus típusát változtatja/változtathatja meg, így értelemszerűen az a típus amelyre konvertálunk mindenképpen referencia vagy pointer kell hogy legyen. Ez leginkább 3 dologra jó, leszármazott típusról bázis típusra konvertálásra (*upcast*), bázisról leszármazottra



A `dynamic_cast` működése.

(*downcast*), valamint többszörös öröklődésnél bázisok közötti váltásra (*sidecast*).
A `dynamic_cast` futási időben végzi a konverziót. Természetesen előfordulhat az, hogy a konverzió nem lehetséges:

```
Sikidom *s = new Sikidom;
Kor *k = dynamic_cast<Kor*>(*s);
```

Ebben az esetben egy hibás downcast-ot csináltunk, hisz `s`-nek a dinamikus típusa is `Sikidom`. Mivel minden `Kor` *az egy* `Sikidom`, de ez fordítva nem igaz, így a konverzió nem lehetséges. Ilyenkor a `dynamic_cast` nullpointert ad vissza. Ha referenciákkal jutunk ilyen helyzetbe, akkor egy `std::bad_cast` típusú kivételt fog dobni.

Tekintsünk példát a `dynamic_cast` működésére:

```
class Base
{
    virtual int f(){} //polimorfizmus szükséges
};
class DerivedOne : virtual public Base {};
class DerivedTwo : virtual public Base {};
class DerivedLast : public DerivedOne, public DerivedTwo {};
//ez a jól ismert gyémántöröklés egy példája

int main()
{
    DerivedLast *dlp = new DerivedLast;
    Base *bp = dynamic_cast<Base*>(dlp); //upcast, dynamic_cast fölösleges
```

```
    DerivedOne *dop = dynamic_cast<DerivedOne*>(bp); //downcast
    DerivedTwo *dtp = dynamic_cast<DerivedTwo*>(dop); //sidecast
    delete dlp;

    bp = new Base;
    dop = dynamic_cast<DerivedOne*>(bp);  //dop nullpointer
    delete bp;

    Base b;
    try
    {
        DerivedOne &dor = dynamic_cast<DerivedOne&>(b);
        //std::bad_cast-ot fog dobni
    }
    catch (std::bad_cast &exc)
    {
        std::cout << exc.what() << std::endl;
    }
}
```

Ez azonban nem hatékony, hisz a `dynamic_cast` futási időben járja végig az öröklődési láncot. Ha nem vagyunk biztosak benne, hogy garantáltan lehetséges a castolás, akkor `dynamic_cast`-ot érdemes használni, ha azonban biztosak vagyunk benne hogy az lehetséges, akkor `static_cast` hatékonyabb.

27.2. static_cast

Amennyiben teljesen biztosak vagyunk abban, hogy a cast szabályos, a `static_cast` is használható mindenhol, ahol a `dynamic_cast` is. Azonban ez az operátor nem végez semmilyen futási idejű ellenőrzést, így sikertelen kaszt esetén a kapott pointer/referencia invalid lesz (használata nem definiált viselkedést eredményez).

A `static_cast` fordítási időben konvertál egy típust egy másik típusra. Annak ellenére, hogy nem feltétlen biztonságos a használata polimorfikus osztályoknál, ez a konverzió végez ellenőrzéseket, méghozzá fordítási időben.

```
//a dynamic_cast-nál látott típusok itt is érvényesek
int main()
{
    Base *bp = new DerivedOne;
    DerivedOne *dop = static_cast<DerivedOne*>(bp); //ok
    DerivedLast *dlp = static_cast<DerivedLast*>(bp); //dlp invalid
    *dlp; //undefined behaviour

    int i = 5;
    double d = 5 / static_cast<double>(i);

    //malloc visszatérési típusa void*
    DerivedTwo *dtp = static_cast<DerivedTwo*>(malloc(sizeof(DerivedTwo)));
    free (dtp);

    Base ba[10];
    bp = static_cast<Base*>(ba);
}
```

E kettő kaszttal közel minden konverziót meg tudunk tenni, melyre szükségünk van és nem okoz nem definiált viselkedést.

27.3. const_cast

A `const_cast` az egyik a két veszélyesebb kaszt közül. Használata csak kivételes esetekben elfogadható, ugyanis nagyon sok galibát okozhat.
A `const_cast` egy konstans objektumról „lekasztolja” a konstansságot:

```
void print(int *param) { printf("%i", *param); }

int main()
{
    const int a = 0;
    print(const_cast<int*>(&a));
}
```

A fenti példa jól demonstrál egy tipikus helyzetet, amikor szabályos a `const_cast` használata. Lévén C-ben nem létezett korábban a `const` kulcsszó, ezért mindent vagy érték szerint vagy nem konstans pointerrel vettek át. Ennek egyik következménye az, hogyha egy régi, C-s függvényt kell igénybe vennünk, mely nem módosítja a paramétert azonban nem konstansként várja, akkor le kell „szednünk” a konstans objektumainkról a konstansságot.

Amennyiben egy nem konstans objektumra konstans referenciával vagy pointerrel hivatkozunk, abban az esetben a nem konstanssá castolt referencián vagy pointeren keresztüli módosítás legális. Amennyiben azonban egy adott objektum konstansként lett deklarálva, annak a módosítása nem definiált.

```
int main()
{
    int i = 3;
    const int& cref_i = i;
    const_cast<int&>(cref_i) = 4; // ok, megváltoztatja i értékét

    const int j = 3;
    int* pj = const_cast<int*>(&j); // maga a cast legális
    *pj = 4; // a módosítás már undefined behaviour
}
```

27.4. reinterpret_cast

Mind közül a legveszélyesebb a `reinterpret_cast`: két tetszőleges típus között végez konverziót, ha értelmezhető az, ha nem. Az értelmes konverziók közül egyetlen eset kivételével minden esetben a fenti 3 cast végre tudja hajtani a konverziót helyesen és sokkal biztonságosabban. Ez azt jelenti, hogy a `reinterpret_cast` használata közel kivétel nélkül elkerülendő.

Az egyetlen konverzió, mely értelmes, és a fentiek nem képesek végrehajtani, amikor polimorfikus osztályoknál egy leszármazottat egy privát ősosztály típusra konvertálunk.

```
class MyVector : private std::vector<int> {};

MyVector *mvp = new MyVector;
std::vector<int> *vp = reinterpret_cast<std::vector<int>*>(mvp);
```

Azonban hiába szabályos ez a konverzió, rengeteg veszélyforrást tartalmaz, hisz a privát öröklés általában épp ennek a megakadályozására szolgál.

27.5. C-szerű cast

A C-szerű castok, vagy más néven *C-style casts* olyan konverziók, melyek a fenti konverziókat próbálják meghívni, a `dynamic_cast` kivételével. Amennyiben az egyik nem sikerül, a következőre haladnak. Ennek az értelemszerű hátránya az, hogy ha mind a `static_cast`, mind a `const_cast` sikertelen, akkor a `reinterpret_cast`-ot fogja használni.

```
int main()
{
    float a = (float)10; //static_cast kerül meghívásra
    const int a;
    int &b = (int&)a; //const_cast
    Base *bp = (int*)b; //reinterpret_cast
}
```

A `reinterpret_cast`-tal ellentétben **soha** sincs szükség erre a cast-ra, csak a C-vel való kompatibilitás miatt része még a nyelvnek.

28. Beugró kérdések

A következő kérdések egy régebbi, kiszivárgott C++ beugróból lettek beemelve. Így értelemszerűen ezekkel a kérdésekkel az olvasó nem fog semmiképpen se találkozni. Alaposabb magyarázatot nem fűztem a megoldásokhoz, ugyanis azok a jegyzet alapos megértése után triviálisak.

1.) Hány byte-on tárol a C++ egy short int-et?

a) 1

b) implementációfüggő

c) 8

d) 2

Válasz: Implementációfüggő.

2.) Melyik reláció **hamis** az alábbiak közül?

a) `sizeof(char) == sizeof(signed char)`

b) `sizeof(short) <= sizeof(long int)`

c) `sizeof(bool) == sizeof(char)`

d) `sizeof(float) <= sizeof(long double)`

Válasz: `bool == char`

3.) Mennyi a 012 konstans értéke?

a) 12

b) 0.12

c) 18

d) 10

Válasz: 10

4.) Melyik nem preprocesszor direktíva?

a) `#define`

b) `#else`

c) `#elif`

d) `#elseif`

Válasz: `#elseif`

5.) Melyik kulcsszó **nem** a tárolási osztályt specifikálja egy deklarációban ill. definícióban?

a) `static`

b) `auto`

c) `register`

d) `public`

Válasz: `public`, ugyanis a register azt jelenti, hogy a változót a `register`be tárolja, azonban azt hogy milyen változót hol érdemes tárolni, jobban tudja a fordító, így nem érdemes kiírni. Az `auto` azt jelenti hogy a változó stacken legyen, ez régebben implicit módon mindehol ott volt, de mostmár nem, sőt, c++11ben mást is jelent.

6.) Az `X::f()` függvényhívás során mit ír ki a program?

```
int i = 1;
namespace X
{
    int i = 2;
    void f()
    {
        int a = i;
        int i = a + X::i + ::i;
        std::cout << i << std::endl;
    }
}
```

a) 1

b) semmit, fordítási hiba keletkezik

c) 5

d) 4

Válasz: 5

7.) Melyik igaz az alábbiak közül?

```
struct X
{
    X(int i = 0) {}
};
```

a) A fenti struct-nak van default konstruktora.

b) A fenti struct-nak csak default konstruktora van.

c) A fenti struct-nak nincs default konstruktora.

d) A fenti struct-nak nincs copy konstruktora.

Válasz: Az első az igaz, továbbá sok egyéb dolgot is generál még.

8.) Az alábbi példában a `Foo f(10);` konstruktor hívása után mennyi lesz `f.x` értéke?

```
struct Foo
{
    int x, y;
    Foo(int i):y(i),x(y++) {}
};
```

a) 11

b) 0

c) nem definiált

d) 10

Válasz: Nem definiált.

9.) Melyik típusnak van push_front tagfüggvénye?

a) `std::set`

b) `std::list`

c) `std::stack`

d) `std::vector`

Válasz: `std::list`

10.) Mi lesz az `a` változó értéke a függvényhívás után?

```
int a = 1, b = 2;
void f(int x, int y)
{
    int t = x;
    x = y;
    y = t;
}
f(a,b);
```

a) 1

b) 2

c) nem definiált

d) semmi, fordítási hiba keletkezik

Válasz: 1

11.) Melyik állítás igaz egy konstans objektum esetében?

a) Az objektumnak csak private adattagja lehet.

b) Az objektumnak csak azok a tagfüggvényei hívhatóak meg, amelyek nem módosítják az adattagjait.

c) Az objektumnak csak a konstans tagfüggvényei hívhatóak meg.

d) Az objektum csak default konstruktorral hozható létre.

Válasz: Az objektumnak csak a konstans tagfüggvényei hívhatók meg.

12.) Melyik állítás igaz az alábbiak közül?

a) A `dynamic_cast` soha nem dob kivételt.

b) A `dynamic_cast` használatához nem lehet statikus adattagja az osztálynak.

c) A `dynamic_cast` használatához polimorf osztályokra van szükség.

d) A `dynamic_cast` fordítás idejű típuskonverziót végez.

Válasz: Polimorf osztályra van szükség a használatához.

13.) Mi a paraméterdedukció?

a) Az az eljárás, amikor referencia-szerinti paraméterátadásra cseréljük az érték-szerintit.

b) Az az eljárás, amikor a fordítóprogram levezeti a template paramétereket a függvényhívásból.

c) Az az eljárás, amikor linker feloldja a külső függvényhívások paramétereit.

d) Az az eljárás, amikor default paraméterekkel látjuk el a függvény paramétereket.

Válasz: Az az eljárás, amikor a fordítóprogram levezeti a template paramétereket a függvényhívásból.

14.) Az alábbiak közül melyik függvényhívással lehet ekvivalens az alábbi (csillaggal jelölt) operátorhívás?

```
class Matrix
{
    // ...
};
Matrix a,b;
a + b; // (*)
```

a) `a.operator+(a,b);`

b) `b.operator+(a);`

c) `operator+(a,b);`

d) `Matrix.operator+(a,b);`

Válasz: A harmadik a jó, mert lehet egy függvényen kívüli függvényhívás a csillagozott rész.

15.) Melyik állítás igaz az alábbiak közül?

a) Egy `int* const` típusú pointer mérete 8 byte.

b) A `sizeof(int) == sizeof(int* const)` reláció mindig igaz.

c) Egy `int* const` típusú pointer nem változtathatja meg a mutatott értéket.

d) Egy `int* const` típusú pointer mutathat változóra.

Válasz: Utolsó.

29. Mintaviszga megoldás

29.1. Az írásbeli vizsga menete

A vizsga írásbeli részénél a feladat egy header fájl elkészítése, mellyel egy előre megadott cpp fájl helyesen fordul. Amennyiben a program szabványos, azaz lefordul és nem okoz nem definiált viselkedést, kiírja a vizsgázó osztályzatát.
A cpp fájl leellenőrzi hogy a header fájlban leírtak helyen oldják meg a feladatot, azonban lehetnek szélsőséges esetek, melyek nem vizsgál. Így a sikeres futás után **sem** biztos, hogy a header fájl a kiírt jegyet megéri.
A kiírt jegy emelett nem „íródik jóvá" azonnal: a vizsgázónak meg kell védenie a munkáját, azaz bizonyítania kell hogy a leírtakat, és a félév során tanultakat valóban érti. Amennyiben erre képtelen, úgy a jegye romlik, szélsőséget esetben elégtelenre is akár.
A header fájl csak is kizárólag a C++98-as szabványnak megfelelő kódot tartalmazhat, ellentkező esetben csak akkor, ha a vizsgázó később meg tudja mutatni, hogy 98-as szabvány szerint mi lenne a megoldás.

29.2. Feladatleírás

A feladat az, hogy írjunk egy osztályt, mely egy billentyűzetet szimulál. Megfigyelhető a lenti C++ kód alapján, hogy `line_editor` lesz a konténer neve, valamint hogy 2 template paraméterrel rendelkezik. Az felhasználónak képesnek kell lennie, arra, hogy a billentyűleütéseit kiirattassa majd a konzolra, sőt, akár arra is, hogy egy adott szövegnek ne csak végére, hanem elejére is írhasson.

lemain.cpp

```
#include <iostream>
#include "lineedit.h"
#include <string>
#include <algorithm>
#include <iterator>
#include "lineedit.h"
#include <list>
#include <vector>

const int max = 1000;

int main()
{
    int your_mark = 1;

    //2-es

    line_editor<std::vector<int>, int> lev;
    for( int i = 0; i < max; ++i )
    {
        lev.press( i );
        lev.home();
        lev.press( i );
    }
```

```
    std::vector<int> v = lev.enter();

    line_editor<std::list<double>, double> lel;
    lel.press( 4.8 );
    lel.home();
    lel.press( 1.1 );
    std::list<double> c = lel.enter();

    line_editor<std::string, char> les;
    les.press( 'W' );
    les.press( 'o' );
    les.press( 'r' );
    les.press( 'l' );
    les.press( 'd' );
    les.home();
    les.press( 'H' );
    les.press( 'e' );
    les.press( 'l' );
    les.press( 'l' );
    les.press( 'o' );

    std::string s = les.enter();
    if ( "HelloWorld" == s && "" == les.enter() && 2.2 > *(c.begin()) &&
    2 * max == v.size() && max - 1 == v[ 0 ] )
    {
        your_mark = c.size();
    }

    //3-as

    les.press( 'H' );
    les.press( 'e' );
    les.press( 'l' );
    les.press( 'l' );
    les.press( 'o' );
    les.home();
    les.insert();
    les.press( 'H' );
    les.press( 'a' );
    s = les.enter();


    for( int i = 0; i < max; ++i )
    {
        lev.press( 2 );
    }
    lev.home();
    lev.insert();
    for( int i = 0; i < max; ++i )
    {
        lev.press( 1 );
    }
    v = lev.enter();

    lel.press( 7.9 );
    lel.press( 1.2 );
    lel.home();
    lel.insert();
```

```
    lel.press( 1.5 );
    lel.insert();
    lel.press( 3.7 );
    c = lel.enter();

    if ( 1.7 > c.front() && 1.3 < c.front() && v[ max / 2 ] == v[ max / 5 ]
       &&
    1.4 > c.back() && 1U * max == v.size() && "Hallo" == s && 1 == v[ max /
       4 ] )
    {
        your_mark = c.size();
    }

    //4-es

    line_editor<std::vector<int> > llev;
    llev.press( 3.3 );
    llev.press( 1.1 );
    llev.home();
    llev.del();
    std::vector<int> lv = llev.enter();

    line_editor<std::string> lles;
    lles.press( 'J' );
    lles.press( 'a' );
    lles.press( 'v' );
    lles.press( 'a' );
    lles.backspace();
    lles.backspace();
    std::string f = lles.enter();

    if ( "Ja" == f && lv[ 0 ] < 1.3 )
    {
        your_mark += lv.size();
    }

    //5-os

    line_editor<std::list<int> > lle;
    lle.press( 3 );
    lle.home();
    lle.insert();

    llev.press( 8 );
    llev.press( 2 );
    llev.home();

    lle.swap( llev );
    lle.press( 1 );
    std::list<int> cl = lle.enter();
    v = llev.enter();
    if ( 1 == cl.front() && cl.size() > v.size() && 3 == v[ 0 ] )
    {
        your_mark += v.size();
    }


    std::cout << "Your mark is " << your_mark;
```

```
    std::endl( std::cout );
}
```

29.2.1. Megjegyzés. Én most kiszedtem a kommenteket, de minden blokk (tehát pl. a 2-es és 3as közötti) ki van kommentezve.

29.3. 1-es

Ahhoz hogy a program egyáltalán lefordúljon, és kiírja hogy `Your mark is 1`, létre kell hoznunk egy header fájlt. Fent látható, hogy ennek milyen nevűnek kell lennie: `lineedit.h`. Az is megfigyelhető, hogy ez a header kapásból kétszer van beillesztve, így (ahogy minden header fájlnál is szokás) írjunk egy header guardot.

```
#ifndef LINE_EDIT_H
#define LINE_EDIT_H

#endif
```

Így már van egy 1esünk.

29.4. 2-es

A ketteshez már létre kell hoznunk magát az osztályt, és ahogy korábban megállapítottuk, 2 template paraméter kell ehhez: az első egy konténer, amiben a felhasználó egy adott sor tartalmát fogja visszakapni, a második pedig a tárolandó típus.

```
template <typename Cont, typename CharT>
class line_editor
{

};
```

A 2-eshez az alábbi függvényeket kell majd megírnunk:

- `press`: A `CharT` típsú kapott paramétert eltárolja.
- `home`: Az adott sor elejére ugrik (nem vár paramétert).
- `enter`: Új sort kezd, és a meglevőt visszaadja egy `Cont` típusú objektumban eltárolva (nem vár paramétert).

Kéne egy konténert választani, melyben eltároljuk a karaktereket. Itt logikus választás az `std::list`, mert gyakran kell majd szúrnunk a sor közepébe.

29.4.1. Megjegyzés. Használhatnánk `std::vector`-t is, sőt bármit, melyet védéskor kellőképpen meg tudunk indokolni.

```
#include <list>

template <typename Cont, typename CharT>
class line_editor
{
    std::list<CharT> line;
};
```

A következő kérdés, hogyan reprezentáljuk a kurzort, mely jelzi hova szúrjon majd be a `press` függvény. Erre praktikus megoldás lehet egy iterátor használata.

```
template <typename Cont, typename CharT>
class line_editor
{
    std::list<CharT> line;
    typename std::list<CharT>::iterator cursor;
};
```

Felmerül az is, kell-e konstruktor, destruktor, copy konstruktor és értékadó operátor?
Ezt az elsőt nem ártana megírnunk: azonban `line`-t és `cursor`-t mivel inicializáljuk? Megállapítható, hogy az `std::list` default konstruktora számunkra megfelelő, így azzal külön foglalkoznunk nem kell. `cursor`-t állítsuk a sor végére.
Mivel mi dinamikusan nem foglaltunk le memóriát, és csak reguláris típusokat tárolunk, így a többi 3 függvényre nincs szükségünk.

```
template <typename Cont, typename CharT>
class line_editor
{
    std::list<CharT> line;
    typename std::list<CharT>::iterator cursor;
public:
    line_editor() : cursor(line.end()) {}

    void press(CharT c)
    {
        //...
    }
    void home()
    {
        //...
    }
    Cont enter()
    {
        //...
    }
};
```

29.4.2. Megjegyzés. Megfigyelhető, hogy `press` érték szerint veszi át a paraméterét – nem kötelező így, de bárhogyan is tesszük, meg kell indokolni. Ebben az esetben lehet rá számítani, hogy a felhasználó csak primitív típusokat akar majd tárolni, legalábbis nem túl komplikáltakat, így az érték szerinti átvétel itt várhatóan hatékonyabb lesz.

A fentiek közül a `home` függvény talán a legegyszerűbb: állítsuk `cursor`-t a sor elejére!

```
void home()
{
    cursor = line.begin();
}
```

Térjünk `press`-re: mivel az iterátor könnyen invalidálódhat ha új elemet szúrunk be, az `insert` az `std::list`-nél visszaad egy iterátort az új elemre, így érdemes a `cursor`-t erre beállítani, és egyel előrébb vinni, hogy a következő beszúrás is az új elem után következzen be.

```
void press(CharT c)
{
    cursor = line.insert(cursor, c);
    cursor++;
}
```

Az `enter`-nél vissza kell adnunk egy, a template paraméterként megadott konténer típususával megyező kontért, mert tartalmazza az eddig begépelt elemeket, majd törli a listát. Mivel minden STL konténer rendelkezik olyan konstruktorral, mely egy iterátor párt vár, és ez alapján a két iterátor közti elemeket be tudja szúrni, így ezt bátran írhatjuk:

```
Cont enter()
{
    Cont ret(line.begin(), line.end());
    line.clear();
    cursor = line.end();
    return ret;
}
```

Ezzel a kettesünk is készen van.

29.5. 3-as

Szükségünk lesz egy `insert` tegfüggvényre, mely ki-be kapcsolja az insert billentyűt (amennyiben ez nem ismerős, az insert meghívásának hatására nem a kurzor után kell majd beszúrni elemeket, hanem a kurzor utáni elemeket kell majd felülírni).
Ehhez szükségünk lesz egy logikai változóra is, melyet alapértelmezetten állítsunk hamisra, és módosítanunk kell majd `press`-t is.

```
template <typename Cont, typename CharT>
class line_editor
{
    //...
    bool isInsert;
public:
    line_editor() : cursor(line.end()), isInsert(false) {}

    void insert()
    {
        isInsert = !isInsert;
    }
    void press(CharT c)
    {
        if(isInsert && cursor != line.end())
            *cursor = c;
        else
            cursor = line.insert(cursor, c);
        cursor++;
    }
    //...
};
```

29.6. 4-es

Megfigyelhetjük, hogy a 4-eshez már a `line_edit` osztályt csak 1 template paraméterrel is tudnunk kell példányosítani. Ez azt jelenti, hogy a második template paraméternek kell egy alapértelmezett behelyettesítést biztonsítani, melynek meg kell egyeznie a `Cont` által tárolt típussal.

```
template <typename Cont, typename CharT = typename Cont::value_type>
class line_editor
{
    //...
};
```

29.6.1. Megjegyzés. A fenti kettős `typename` talán összezavaró lehet – a másodikra a dependent scope miatt van szükség.

29.6.2. Megjegyzés. Mivel internetet lehet használmi, nem kell megijedni, ha nem tudjuk ezt hogyan kéne egyből megcsinálni. Előfordulhat olyan, hogy olyan dologgal találkozunk vizsgán szembe, amit se előadáson, se gyakorlaton nem mondtak el: ez azért van ,mert ennek a tárgynak célja az, hogy ezekkel is meg tudjunk küzdeni.

Ezen kívül két új tagfüggvényt is meg kéne írnunk:

1. `backspace`: Az `cursor` mögötti elemet törli ki.
2. `del`: A `cursor` előtti elemet törli.

```
template <typename Cont, typename CharT = typename Cont::value_type>
class line_editor
```

```
{
    //...
public:
    void backspace()
    {
        cursor--;
        del();
    }
    void del()
    {
        cursor = line.erase(cursor);
    }
    //...
};
```

29.7. 5-ös

Meg kell írnunk egy swap függvényt, mely 2, különböző template paraméterrel rendelkező `line_editor`-t megcserél.

```
//...
#include <algorithm>

class line_editor
{
    //...
public:
    //...
    template <class T, class U>
    void swap(line_editor<T,U> &le)
    {
        std::swap(line, le.line);
        std::swap(cursor, le.cursor);
        std::swap(isInsert, le.isInsert);
    }
};
```

Azonban fordítási hibát kapunk, hisz egy különböző template pareméterekkel rendelkező `line_editor` külön típusnak számít, így gondoskodunk kell arról is, hogy minden `line_editor` barát legyen.

```
class line_editor
{
    //...
public:
    //...
    template <class T, class U>
    friend class line_editor;
};
```

Így az ötöst is elértük.