C++ jegyzet

2023/2024/1. félév

Pataki Norbert előadásai és
Brunner Tibor gyakorlatai alapján

Utolsó módosítás: 2024. január 28.

Tartalomjegyzék

Előszó

Ezt a jegyzetet a 2023/2024. őszi félév végén állítottam össze azzal a szándékkal, hogy összegyűjtsem és rendszerezzem az órákon elhangzott ismereteket. A tantárgy nagy hangsúlyt fektet arra, hogy ne csak gyakorlati szempontból tudjuk használni a C++ eszköztárát, hanem (meg)értsük, átlássuk a belső működését, a létrejöttének hátterét. Éppen ezért ez a jegyzet a vizsga elméleti részére való felkészülésben hivatott segítséget nyújtani.

Jelen jegyzet esetében a szó szoros értelmében jegyzetről van szó, azaz a tanulást segíti, az anyagot áttekinthetőbbé, rendszerezettebbé teszi, de nem ajánlom, hogy pusztán ebből készüljünk fel. Inkább kiegészítő anyagnak javaslom a használatát.

A jegyzet elkészítéséhez felhasználtam az előadás diasorait, a gyakorlaton írt jegyzeteimet, valamint Umann Kristóf jegyzetét (`https://people.inf.elte.hu/szelethus/LaTeX/cpp/cpp_book/cpp_book.pdf`). Sok segítséget jelentett *The Cherno* YouTube csatornája a definíciók megértésében.

Igyekeztem a legjobb tudásom szerint megírni, azonban így is lehetnek benne hibák, pontatlanságok, elgépelések. Ha találsz ilyet, kérlek értesíts az alábbi címen: ap3558@inf.elte.hu.

A gyakorlati részére korábbi évek vizsgafeladataival lehet felkészülni, amelyeket az előadó, Pataki Norbert honlapján lehet megtalálni: `http://patakino.web.elte.hu/pny2/`

Sikeres felkészülést kívánok!

Nimród

1. fejezet

C++ alapismeretek

1.1. Motiváció

- A '70-es években megjelent a **C programozási nyelv**, amely nagy sikernek örvendett.
  - statikusan típusos, imperatív-procedurális paradigmájú nyelv
  - anno elég magas szintűnek számított (egy Assemblyhez képest mindenképp), ami kényelmessé tette a megtanulását
  - hatékonyság; pl. a szabvány a típusoknak a méretét nem korlátozza[1], hanem a fordítóra bízza az adott architektúrának megfelelő kiválasztását
- Azonban voltak (vannak) limitációi is.
  - **Konstansság** kijátszásának lehetősége
  - **Osztályok hiánya**: léteznek `struct`ok, melyekhez írhatunk *függvények*et, de az adattagok elfedését nem támogatja, hiszen minden publikus
  - **Generikusok hiánya**: legközelebbi megoldások a makrók és a `void*`-ek használata
    - Makrók – fordítási időben nem (vagy nagyon nehezen) ellenőrizhetők, nem ismerik a típusát az argumentum(ok)nak, hiszen egyszerű szövegbehelyettesítésről van szó
    - `void`-pointerek – kasztolás során elvész a típusinformáció, ha nem vagyunk óvatosak
  - **Függvények nevei**: C-ben a függvényeket a nevük határozzák meg. Ez gondot jelent, mert ha két adatszerkezet rendelkezik ugyanolyan nevű művelettel, akkor mindkettőnek egyedi nevet kell biztosítanunk. Más könyvtárak más elnevezési konvenciókat követnek, így nagyobb projektek esetén nehézkessé válik a nyomonkövetésük.
- **Bjarne Stroustrup**, dán informatikus és számítógéptudós, a '80-as években elkészítette a C++ nyelvet, ami ezekre a hiányosságokra próbál megoldást nyújtani.

[1] Kivétel ezalól a `char`, mivel annak az esetében `sizeof(char) == 1`.

  - Maradjon ugyanolyan hatékony, mint a C, de magas-szintű nyelvi eszközöket támogasson, például:
    - a konstansság védelmére új eszközök (**referencia** szerinti paraméterátadás, `const` kulcsszó új kontextusai),
    - objektumorientált programozás: **osztályok, származtatás, adatelrejtés, enkapszuláció**,
    - típusfüggetlen algoritmusok a **sablonok** (*template*-ek) segítségével,
    - **névterek** bevezetése, a függvényeket az „összekutyult nevük" (*mangled name*) és a paraméterlistájuk határozza meg.
  - És mindezek mellett legyen **visszafele kompatibilis már létező C kóddal** (már amennyire azt ésszerű támogatni).

1.2. A C++ fordítómodellje

1.2.1. A fordítás eszközei

Fordítóprogramok: `g++`[2], `clang`, Microsoft Visual C++ (MSVC) compiler, stb.

Forrásfájljai: fordítási egység (`*.cpp`), header fájlok (`*.h`, ritkábban `*.hpp`)

Valójában a fordító (*compiler*) nem is nézi a bemeneti forráskód fájlkiterjesztését, akár `*.txt` fájlt is lefordít, ha az éppenséggel C++ kódot tartalmaz.

Szétválasztjuk a függvények deklarációját a definíciótól, azaz az előbbit a header fájlban, míg az utóbbit a fordítási egységben írjuk meg.

A header fájlokban használjunk **header guard**okat – szabványos módon. Ezzel a linkelési hibákat tudjuk csökkenteni, valamint modularizálni tudjuk a kódunkat.

```
1 #ifndef HEADER_GUARD_H
2 #define HEADER_GUARD_H
3
4 // rest of the code
5
6 #endif // HEADER_GUARD_H
```

Szabványos header guard

Az alábbi megoldás *nem szabványos*, ugyanis nem része a szabványnak. Tehát, nem várható el minden fordítótól, hogy támogassa.

```
1 #pragma once
2 // rest of the code
```

Nem-szabványos header guard

A *modularizáció* javítja a programkód átláthatóságát, karbantarthatóságát, fokozza a csapatmunkát. Megengedi, hogy csak azokat fordítsuk újra, amiket módosítottunk (a teljes program helyett).

[2] Lényegében megegyezik a `gcc`-vel, leszámítva, hogy alapértelmezetten C++-t fordít, de ezt egy kapcsolóval módosítani tudjuk. Hasonlóan, a `gcc`-vel is tudunk C++ kódot fordítani a megfelelő *compiler flag*gel.

1.2.2. A C++ fordító fázisai

I.) Preprocesszálás

- egyszerű szöveges *search-and-replace* (kivágás, beszúrás, stb.) módosításokat végez a bemeneti forráskódon → lassú folyamat
- **nyelvfüggetlen**, nem ismer semmilyen C/C++ nyelvi szabályt[3]
- az alábbi **preprocesszor direktívák**at támogatja (C++11-ig bezárólag): `#if`, `#else`, `#elif`, `#endif`, `#include`, `#define`, `#undef`, `#ifdef`, `#ifndef`
- fontos, hogy ezek a **fordítás előtt** történnek meg, azaz egy olyan forráskódot állít elő, ami készen áll a nyelvi fordításra
- konvencionálisan csupa nagybetűkkel nevezzük el ezeket a makrókat
- különlegessége, hogy makrófüggvényeket is definiálhatunk
  - C-ből származó örökség
  - mivel nyelvfüggetlen, nem ismeri az argumentumok típusát
  - a változók nevei bekavarhatnak, ezért indokolt körbezárójeleznünk őket a makró kódjában
  - kerüljük a használatukat C++-ban, hiszen rendelkezünk sokkal biztonságosabb eszközökkel

```
#include <iostream>
#define SQUARE(x) ((x)*(x))
int main() {
    int i = 5;
    int e01 = SQUARE(i);
    int e02 = SQUARE(2+3);
    int e03 = SQUARE(i++);
    std::cout<< e01 << " " << e02 << " " << e03 <<
        std::endl;
    // result: 25 25 30 (30 == 5 * 6) (compiled in g++)
    return 0;
} // results may vary depending on the compiler
```

- van lehetőség C/C++ **feltételes fordítás**ra, azaz C-ben írt header fájlt C++ kódban felhasználni és C++ fordítóval lefordítani, **előredefiniált preprocesszor szimbólumok** segítségével; az adott build környezet szerint fog működni másképp a program

```
#ifdef __cplusplus
extern "C" {
#endif
// ...
#ifdef __cplusplus
}
#endif
```

[3] Ellenben megoldható, hogy olyan nyelvekben is végrehajtsunk preprocesszálást, amik eleve nem támogatnak preprocesszor direktívákat (pl. Java).

II.) Nyelvi fordítás

- forráskód → gépi (bináris) kód
- fordítási ellenőrzéseket végez, figyelmeztetéseket (akár hibaüzeneteket) is dob
  - lexikális egységekre bontja fel a forráskódot (*tokenizáció*)
  - meghatározza, hogy milyen szerepet töltenek be a tokenek (*lexikális elemzés*) (lényegében: mi a szófaja?)
    - **kulcsszók**: `for`, `int`, `virtual`, `static`, `namespace`, `const`, ...
    - **azonosítók**: `std`, `cout`, `Foo`, `foo`, `_i`, `vector`, `string`
      - az azonosítók *case-sensitive*-ek
      - fontos, hogy jó azonosítónevet használjunk
      - konvenciók: ***snake case*** (`snake_case`), *camel case* (`CamelCase`), *Hungarian notation* (a változó nevébe bele van égetve a típusa, mára elavult szokás)
      - kerüljük az ékezeteket... meg általában a nem-angol elnevezéseket
      - a név elején lévő aláhúzásjel / alsókötőjel (*underscore*) sem jó ötlet, ugyanis ezek fenntartottak
    - **konstans szövegliterálok**: `"hello␣world"`
      - a C-vel való visszafele-kompatibilitás csomó gondot okoz
      - a C-ben a fenti példa típusa: `char[12]` (nem pointer, tömb)
      - C++-ban a fenti példa típusa: `const char[12]` (szintén nem pointer)
      - viszont ha a tömb pointerré kasztolódik, akkor elvész a tömb mérete (a `sizeof` nem az elvárt eredményt fogja adni)
      - továbbra is fontos, hogy `'\0'` szerepeljen a végén
      - bevezették az `std::string` típust, ami sok terhet levesz a vállunkról
    - **számliterálok**: `123`, `12.34`, `0x123`, `42UL`, stb.
    - **operátorok**: `<<`, `+`, stb.
      - precedencia: melyik részkifejezés értékelődik ki hamarabb
      - asszociativitás: balra- vagy jobbra köt
      - aritás: operandusainak száma (unáris, bináris, ternáris)
      - fixitás: prefix, infix, postfix
      - típus: pl. az osztás másképp működik `int` és `double` esetében
      - mellékhatás, eredmény: C/C++-ban meghatározzák az operátort
    - **szeparátorok**: `';'`, `','`, `'\t'`, `'␣'`[4] (*whitespace* karakterek)

[4] Ez a szóközt hivatott szemléletesebben ábrázolni.

  - ebből felépíti a program **absztrakt szintaxisfá**ját a kifejezésekből
  - ezután következik a *szemantikai elemzés* (⇔ van-e értelme annak, amit leírtunk?)
- optimalizál(hat)ja a kódot
  - egy tipikus példa rá: **return value optimisation** (RVO)
  - vegyük az alábbi példát: *hány objektum jön létre*?

```
1  struct S {};
2
3  S f() {
4      S s;          // 1 object has been created
5      return s;
6  }
7
8  int main() {
9      S s = f();    // copy constructor is executed
10     return 0;
11 }
```

  - a nyelv szabályai szerint 2 jön létre → de ez ellentmond a C++ hatékonyság iránti elköteleződésének
  - a compiler ezt úgy optimalizálja, hogy csak 1 objektum jön létre
  - nem figyelmezteti róla a programozót, annak ellenére, hogy egy **nagyon erős optimalizáció**ról van szó

III.) Összeszerkesztés (linkelés)

1.3. Típusrendszer

- a **processzor**ban regiszterek vannak
  - **regiszter**: ideiglenes tárolóegység, ami betölt n bitet a memóriából, gyors számításokat végez rajta, majd visszahelyezi
    - 64-bites processzor ⟺ 1 regiszter 64 bitet képes tárolni
    - jó néhány **általános célú regiszter**
      - 64-biten: RAX, RBX, ...
      - 32-biten: EAX, EBX, ...
      - 16-biten: AX, BX, ...; 8-biten: AH, AL
      - backward kompatibilitás a régebbi architektúrákkal
    - **lebegőpontos regiszterek** – sokáig nem volt általános a hardveres támogatottságuk
    - egyéb **specifikus regiszterek**
  - maga a regiszter **típustalan**, bármit tárolhat, ami belefér (egész szám, struct, memóriacím, koordináta)

- a **típus** az a programozási nyelvekben létező **konstrukció, ami egy bitsorozathoz valamilyen jelentést társít**
  - a jelentés meghatározza, milyen értelmes műveleteket végezhetünk el rajta
  - fordítási visszajelzések, pl. `strlen()`-nel ne kérdezzük le egy `double` hosszát, mégha bele is fér 64-bitbe
- `int` – **alapértelmezett számábrázolás**
  - **előjeles, kettes komplemensű számábrázolás**: a bitsorozat elején lévő bit(ek) felelős(ek) az előjel (vagy helyiérték) jelöléséért
  - `sizeof(int)` → implementációfüggő
  - `unsigned` számábrázolás esetén az előjelbitet **helyiértékbit**ként használja fel
  - a konstansliterálokon jelezhetjük szuffixumokkal, hogy pontosan milyen típusú
    - `42L` – `long`
    - `42U` – `unsigned int`
    - `42UL` – `unsigned long`
- `char` – **egységtípus**a a nyelvnek
  - azaz `sizeof(char) == 1` → ezt a szabvány garantálja
  - minden más típust egyenlőtlenségekkel határoz meg ($\leq$ relációval)
  - a `char` előjelessége implementációfüggő
  - `sizeof(char)` $\leq$ `sizeof(short)` $\leq$ `sizeof(int)` $\leq$ `sizeof(long)`
  - `sizeof(bool)` $\leq$ `sizeof(long)` – a C-s hagyomány és visszafele kompatibilitás megőrzése miatt esett erre a választás
- `double` – **alapértelmezett lebegőpontos számábrázolás**
  - a `float` fele annyi bittel írja le azt, amit a `double`, ezért fele olyan precíz
  - jellemzően **nem a tartomány a szempont, hanem a pontosság**
  - szuffixolás, alternatív jelölések
    - `12.34` – alapértelmezetten `double`
    - `12.34f` – `float`
    - `12.34L` – `long double`
    - `88e-1` $\iff 88 \cdot 10^{-1}$ ([előjel], mantissza, exponens) $\iff 8.8$
    - `1234e-2f` – 12.34 (`float`)
    - `54e-1L` – 5.4 (`long double`)
  - IEEE 754, lebegőpontos számábrázolási szabvány
  - `sizeof(char)` $\leq$ `sizeof(float)` $\leq$ `sizeof(double)` $\leq$ `sizeof(long double)`
  - a lebegőpontos számábrázolás pontatlanságait tartsuk szemelőtt!

1.4. Deklaráció, definíció

Deklaráció

Egy névvel rendelkező entitáshoz típust rendelünk. Így beszélhetünk *változódeklaráció*ról és *függvénydeklaráció*ról.

Definíció

Változó esetén: meghatározzuk magát a változót, **tárhely foglalása történik**.
Függvény esetén: megadjuk a **függvénytörzs**ét.
Kapcsolódó fogalom: **one definition rule** (ODR)

Inicializáció

Egy változónál a **legelső értékadást** jelenti.

```
#include "mylib.h" // mylib.h: int global_var;

extern int global_var; // only declaration, not definition

// forward declaration
struct MyStruct;
class MyClass;

int func(int x, int y); // declaration

int main()
{
    int i;       // declaration, definition
    int j = 42; // declaration, definition and initialisation
    return 0;
}

// definition of the previously declared function
int func(int x, int y)
{
    return x + y;
}
```

Egy összefoglaló példa deklarációra, definícióra és inicializációra

1.5. Láthatóság, elfedés

Blokkutasítás: `{ ... }` között elhelyezett utasítássorozat.

Lokális változó: blokkutasításon belül deklarált változó.

Blokkra nézve lokális: abban a blokkban van, amiben vizsgáljuk.

Blokkra nézve nonlokális: a külső (bennfoglaló) blokkban van, de az aktuális blokk a deklaráció hatókörében van.

Globális változó: nem tartozik semmilyen blokkhoz.

Elfedés (*shadowing, hiding*): egy belső blokkban ugyanolyan nevű változót deklarálunk, amilyen a blokkon kívül már deklarálva van. Ilyenkor a belső elfedi a külső nevét, így nem sérül a *one definition rule*.

```
#include <iostream>

int main()
{
    int n = 0;

    { // block starts here
        std::cout << n << std::endl; // 0
        int n = 1;
        std::cout << n << std::endl; // 1
    }

    std::cout << n << std::endl;     // 0

    return 0;
}
```

1.6. A C++ memóriamodellje

Három részből áll a C++ memóriamodellje (ahogyan a C-nek is).

I.) Stack:

- **miket tárol**: lokális változók, függvények aktivációs rekordja → *automatikus változók*
- verem adatszerkezetben
- **élettartam**: definiálástól a blokk végéig tart (ahol lefut a destruktoruk)
- a destruálás fordított sorrendben történik a konstruáláshoz képest

II.) Heap:

- **miket tárol**: mindent, amit a `new`, `new[]` operátorokkal példányosítunk → *dinamikus változók*
- **élettartam**: memóriafoglalástól (`new`, `new[]`) felszabadításig (`delete`, `delete[]`) (NEM fut le a destruktor automatikusan a blokk végén)
- potenciális problémák:
  - elfelejtjük felszabadítani → memóriaszivárgás
  - felszabadítás után hozzáférés (*dangling pointers*) → olyan memóriaterületet akarunk elérni, amihez (már) nincs jogunk hozzáférni → szegmentációs hiba
  - kétszer szabadítjuk fel a területet → definiálatlan viselkedés

III.) Statikus tárterület:

- **miket tárol**: globális változók, statikus függvények, sztringliterálok
- **élettartam**: a program elejétől a végéig

1.7. Névterek

A **C-ben** problémát jelentett, hogy a függvényt a nevük alapján azonosítja be a fordító, emiatt **minden függvénynek egyedi névvel kell rendelkeznie**. Ha sok adatszerkezetet használunk, melyek azonos nevű, hasonló műveletekkel rendelkeznek (pl. `add()`), akkor mindegyiknek egyedi `add`-függvényt kéne írnunk. Ez kényelmetlenné teszi a kód áttekinthetőségét és karbantarthatóságát – nem beszélve arról, hogy más könyvtárak más elvenezési konvenciókat használhatnak, így a kódunk még kaotikusabbnak fog kinézni.

Erre a problémára nyújtanak megoldást a **névterek**. Egy névtér úgymond „becsomagolja" a benne definiált függvényeket, típusokat, stb., így egy „közös csatlakozási pontból" hivatkozni tudunk rájuk – ezzel **elkerülve a névütközéseket**.

Használata végtelenül egyszerű; a névteret a `namespace` kulcsszóval tudjuk létrehozni és amit a blokkjába írunk, az abba fog tartozni. Bármit tartalmazhat: **változót, osztályt, függvényt**[5]. A hivatkozást a `::` operátorral tehetjük meg.

Létezik egy ún. **névtelen** vagy **anonim névtér**. Függvények esetében szinte megegyezik azzal, ha `static` kulcsszóval látnánk el az elején. Tehát **minden, amit az anonim névtérbe írunk, az a fordítási egységre nézre lokális marad**. Hatalmas előnye, hogy így lokális típusokat is tudunk definiálni – amit egy sima `static` kulcszóval nem tudnánk elérni.

```
1  namespace Namespace01
2  {
3  struct Struct01 {};
4  int return_value() { return 0; }
5  }
6
7  namespace Namespace02
8  {
9  struct Struct02 {};
10 int return_value() { return 42; }
11 }
12
13 // anonymous namespace
14 namespace
15 {
16 struct Struct03 {}; // S03 remains local in the translation unit
17 int return_value() { return -1; }
18 }
19
20 int main() {
21     int v01 = Namespace01::return_value(); // 0
22     int v02 = Namespace02::return_value(); // 42
23     int v03 = ::return_value(); // -1
24 }
```

A **C++** emellett a megváltoztatta a függvények beazonosítására vonatkozó szabályokat. Egy függvényt több tulajdonsága határoz meg: **a paraméterlistája, a neve, a névtere, visszatérési értéke**. Ezeket az információkat egy „*összekutyult név*ben", avagy **mangled name** gyúrja össze. Ez az a név, amit a fordító generál és a binárisban használatos.

[5] Ez később vissza fog térni, ugyanis egy súlyos probléma okozója lesz (ld. *Sablonok, Iterátorok*).

1.8. Referenciák

A C-ben csak érték szerinti paraméterátadás létezik. Ha pointert adunk át, valójában az adott memóriacím másolódik le.

A **pointerek** ugyanúgy léteznek C++-ban is, azonban rendelkeznek azzal a súlyos hibával – amit a C-től örökölt –, hogy **könnyen megsérthetjük a konstansságát a mutatott változónak**. Többek között erre hivatott megoldást nyújtani a referencia.

- A referenciatípust a `&` karakterrel jelöljük. Vigyázat: ez nem a címlekérdező operátor!
- A referencia egy már definiált változónak a fedőneve (egyfajta *alias*). Nem jön létre, ha nem létezik olyan objektum, amire referáljon – magyarul: **inicializálni kötelező**.
- Inicilizációt követően **nem állíthatjuk át, hogy más objektumra hivatkozzon** (ellentétben egy pointerrel). A példában szereplő művelet azt sugallja, mintha a `reference` változót átállítanánk úgy, hogy a `temp`re hivatkozzon, de valójában csak értékül adja a `temp` értékét.
- Amikor függvényparaméterként adjuk át, akkor **nem másolódik le a teljes objektum**, hanem a függvényből tudjuk **módosítani az eredeti objektum értékét**.
- Ugyan a háttérben egy pointerről van szó szintaktikai mázzal, **nem** adhatunk értékül neki `NULL` értéket (vagy C++11-től `nullptr` értéket). Hasonlóan, **nem használ pointeraritmetikát**, tehát nem kell dereferálnunk, címet lekérdeznünk. Összetett típusoknál a `.` adattaglekédező operátort használjuk a `->` operátor helyett.
- **Konstans referencia** átadása esetében se nem állíthatjuk át, hogy melyik objektumra referáljon, se nem módosíthatjuk a belső állapotát – így biztonságosabb a pointereknél.
- Konstans referenciának értékül adhatunk jobbértéket. Sima referenciának nem.

```
#include <iostream>
// passing argument as constant reference
void print_variable(const int& var) {
    std::cout << var << std::endl; // no need to dereference it
}
int main() {
    int var = 42;
    int temp = 5;

    int& reference = var; // initialisation is obligatory!
    reference = temp;     // equivalent to: var = temp;

    print_variable(var);  // no need to use the '&' operator
                          // result: 5
}
```

1.9. Balérték, jobbérték

Balérték (*lvalue*): egy címképezhető (`&`) objektum.

Jobbérték (*rvalue*): ami nem balérték. Ezek jellemzően literálok (`42`, `"text"`), érték szerint visszatérő függvény visszatérési értéke. Pl. annak, hogy `&5` nincs értelme, mert az `5` nem egy memóriában tárolt változó.

2. fejezet

Objektumorientált programozás

Objektumorientált programozási nyelv

Egy programozási nyelvet **objektumorientált**nak nevezünk, ha az alábbi eszközöket támogatja: **enkapszuláció, adatelrejtés elve, öröklődés** (származtatás).

A C++-ban saját típust a `class` és a `struct` kulcsszókkal lehet deklarálni / definiálni. A kettő között a különbség annyi, hogy a `class`ban alapértelmezetten az adattagok privát láthatóságúak, a `struct`ban publikusak. Minden másban teljesen egyformán viselkednek.

A C++ az osztályok műveleteit tagfüggvényeknek hívja, ellentétben a metódussal, amivel OEP-en megismerkedtünk.

Mekkora egy osztály / struktúra mérete? Akkora, amekkora az adattagjainak méretének összege, plusz még egy kevés (hogy mennyivel, az implementációfüggő, de kikapcsolható). Tehát a tagfüggvényeinek pointerei nem számítódnak bele.

Az az oka, amiért a két konstrukció szinte teljesen megegyezik, mert a `struct`ot a C-ből örökölte, ezért visszafele kompatibilisnak kell maradnia vele (ezért alapértelmezetten nyilvánosak az adattagok); míg a `class` elnevezést meg a Simula 67-től vette át[1], mivel mire a C++ megjelent, addigra konvencióvá vált az elnevezés.

2.1. Fordító által generált tagfüggvények

2.1.1. Alapértelmezett konstruktor

A C++-ban minden típusnak létezik konstruktora. Még a primitív típusoknak is.

```
int i(42); // equivalent to: 'int i = 42;' or 'int i = int(42);'
int j(i);  // equivalent to: 'int j = i;' or 'int j = int(i);'
```

Osztályban az alábbi módon írhatunk konstruktort.

```
class Point {
private:        // this line can be omitted
    int x, y;
};
```

[1] Simula 67, az első objektumelvű programozási nyelv.

És a vicc az, hogy explicite nem is kell írnunk semmit, hiszen ilyenkor a fordító generál egy ún. **alapértelmezett konstruktor**t (*default constructor*).

Alapértelmezett konstruktor

Az alapértelmezett konstruktor egy **paraméter nélküli** konstruktor. Ezt generálhatja a compiler, de mi magunk is írhatunk ilyet.

Az alábbi módon írhatunk olyan konstruktort, ami egyszerre elfogad 0, 1 vagy 2 paramétert is, ugyanis ha nem töltjük ki, akkor a megadott értéket fogja hozzárendelni az adattaghoz.

Vigyázat: itt a `this` egy `Point` típusú mutató, ami pontosan az inicializálandó objektumra mutat. Tehát nem kulcsszóról van szó (mint Javában, C#-ban).

```
class Point
{
private:
    int x, y;

public:
    Point(int x = 0, int y = 0)
    {
        this->x = x;
        this->y = y;
    }
};
```

Ezt akár még tömöríthetjük is tovább. A kettőspont után az **inicializáló listá**val a korábban megismert módon tudjuk inicializálni az adattagokat. Az, hogy a nevek megegyeznek, nem jelent gondot, hisz a külső az osztály adattagjára, a belső a konstruktor paramétereire vonatkozik. **Az inicializáló listában a deklaráció sorrendjében történik az inicializáció, nem a listabeli felsorolásában.**

```
class Point
{
private:
    int x, y;

public:
                                    // initialiser list
    Point(int x = 0, int y = 0) : x(x), y(y) { }
};
```

Minden adattagját az osztálynak a konstruktorban kell inicializálni (C++11-ig).

2.1.2. Destruktor

Vegyünk egy másik példát. Írjunk egy olyan tömb osztályt, ami n darab `int`et tárol, és lekérdezhető a mérete futási időben. Rendelkezzen `[]` operátorral. Érjük el azt, hogy lehessen inicializálni egy tömböt úgy, hogy a forrástömböt az egyenlőségjel túloldalára helyezzük.

```
Array a(5);
for (int i = 0; i < a.capacity(); i++) a[i] = 42;
Array b = a; // initialisation with the contents of 'a'
```

Az osztály alapja valahogy így nézne ki.

```
class Array {
private:
    int* data;
    int  capacity;

public:
    Array(int capacity) {
        // minimal error handling
        if (capacity == 0) capacity  = 1;
        if (capacity < 0)  capacity *= (-1);

        this->data = new int[capacity];
        this->capacity = capacity;
    }
};
```

Először győződjünk meg róla, hogy működik-e. Legyen a `main()` függvényünk az alábbi:

```
// definitions of Point and Array
int main() {
    Point p;
    Array a(5);
    return 0;
}
```

Ellenőrizzük a `valgrind`dal, hogy nem szivárog-e memória.

```
==11158== Memcheck, a memory error detector
==11158== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==11158== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
==11158== Command: ./pointarray
==11158==
==11158==
==11158== HEAP SUMMARY:
==11158==     in use at exit: 20 bytes in 1 blocks
==11158==   total heap usage: 2 allocs, 1 frees, 72,724 bytes allocated
==11158==
==11158== LEAK SUMMARY:
==11158==    definitely lost: 20 bytes in 1 blocks
==11158==    indirectly lost: 0 bytes in 0 blocks
==11158==      possibly lost: 0 bytes in 0 blocks
==11158==    still reachable: 0 bytes in 0 blocks
==11158==         suppressed: 0 bytes in 0 blocks
==11158== Rerun with --leak-check=full to see details of leaked memory
==11158==
==11158== For lists of detected and suppressed errors, rerun with: -s
==11158== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Sajnos azt tapasztaljuk, hogy igenis szivárog. Ami viszont érdekesebb, hogy miért csak az egyik?

Idézzük fel az automatikus változókról tanultakat! A blokk legvégén a destruktoruk lefut automatikusan – de mi nem írtunk destruktort egyikhez sem. Ez így van, viszont a fordító generál nekünk egyet. Ha újból megnézzük, hány bájt veszett el (20 bájt = `sizeof(int) * 5` a gépemen), feltűnhet, hogy pontosan a tömb méretének megfelelő tárterület szivárgott el – nem csoda, hisz az egy dinamikus változó, azt nekünk kell destruálnunk.

Röviden, a destruktor is egy olyan tagfüggvény, amit **legenerál a fordító**. Összesen négy ilyen tagfüggvény van: az **alapértelmezett (default) konstruktor**, a **destruktor**, a **másoló (copy) konstruktor** és az **értékadó operátor**.

Azonban, ha *erőforrásokkal dolgozik az osztályunk* (fájllal, allokált tárterülettel), akkor életbe lép a **hármas szabály (rule of three)**.

Rule of three

Ha az osztályunk erőforrásokkal dolgozik és legalább az egyik, a fordító által generált tagfüggvény nem megfelelően működik, akkor írjuk meg mind a hármat saját magunk (feltéve, hogy a konstruktort már megírtuk).

Szerencsére a destruktornak nem bonyolult a szintaxisa, így megírhatjuk könnyedén az `Array` osztálynak. A `Point`nak szükségtelen, hisz az nem foglal le dinamikusan memóriát.

```
1  class Array
2  {
3  // ...
4  public:
5      // constructor
6      Array(int capacity) { ... }
7
8      // destructor
9      ~Array() { delete[] this->data; }
10 };
```

2.1.3. Copy konstruktor

Korábban megbeszéltük, hogy a fordító generál nekünk másoló konstruktort is. De pontosan mi is a feladata?

Copy konstruktor

Egy copy konstruktor egy olyan konstruktor, amit egy **ugyanolyan osztálytípusú argumentummal hívunk és átmásolja az argumentuma tartalmát anélkül, hogy azt módosítaná**.
A copy konstruktor akkor hívódik meg, amikor egy *másik ugyanolyan típusú objektummal* **inicializálják** [...].[a]

```
1 T t01;            // default constructor
2                   // equivalent to: T t01 = T();
3
4 T t02 = t01;      // copy constructor
5                   // equivalent to: T t02(t1);
```

[a] A definíció eredeti angol változata: `https://en.cppreference.com/w/cpp/language/copy_constructor`

Megkülönböztetünk kétfajta másolást a programozásban.

1. **Sekély másolás** (*shallow copy*): amikor az adatszerkezetet adattagonként másoljuk le. Heapen allokált objektumok esetén ez a memóriacímek lemásolását jelenti.
   Ez a legtöbb esetben nem jó nekünk, hiszen ha a forrásobjektumot destruáljuk, az felszabadítja azt az erőforrást is (pl. deallokálja a tömböt), amire a másolat mutat. Így ha a másolatot használnánk a destrukció után, dereferálni fogjuk, ami szegmentációs hibát fog eredményezni.
2. **Mély másolás** (*deep copy*): a heapen allokált erőforrásokból új példányt készít a másolat számára. Így ugyanazon adathalmazból valóban még egy példány fog létrejönni a memóriában.

Elevenítsük fel, hogy a copy konstruktor **inicializáció** során fut le, ami csak egyszer fog bekövetkezni az objektum élettartama során. Tehát nem kell deallokálnunk semmit.

```
1  class Array {
2      // ...
3  public:
4      // ctor, dtor
5
6          // copy ctor
7          Array(const Array& other) : capacity(other.capacity) {
8              data = new int[capacity];
9
10             for (int i = 0; i < capacity; i++) {
11                 data[i] = other.data[i];
12             }
13         }
14 };
```

2.1.4. Értékadó operátor

Értékadó operátorból *több létezik*, ezt a primitív típusoknál is tapasztalhattuk (=, +=, stb.). Általános jellemzőik, hogy módosítják azon objektum értékét, amire meghívjuk. Az egyszerű értékadó operátort a szabvány **másoló értékadó operátor**nak (*copy assignment operator*) nevezi. Mi azonban ezt egyszerűen *az* értékadó operátornak fogjuk hívni.

Az értékadó operátor abban az esetben lép életbe, ha **van két deklarált és inicializált objektumunk, és az egyiket értékül akarjuk adni a másiknak**. A célunk, hogy *mély másolás*t hajtson végre az operátor, viszont ügyelnünk kell arra, hogy **felszabadítsuk a heapen allokált tárhelyet** szükség esetén.

Értékadó operátor

Az értékadó operátorok **módosítják az objektum értékét**.
Minden beépített értékadó operátor **`*this`sszel tér vissza** és a legtöbb felhasználó által definiált túlterhelés is ezzel tér vissza, hogy ugyanúgy lehessen használni őket, mint a beépítetteket.[a]

```
T t01, t02;       // default constructor
                  // equivalent to: T t01 = T(); T t02 = T();

t02 = t01;        // assignment operator
                  // equivalent to: t02.operator=(t01);
```

[a] A definíció eredeti angol változata: `https://en.cppreference.com/w/cpp/language/operator_assignment`

Ne feledjük, hogy az operátor egy `Array` típusú **referenciával tér vissza**, ezért a mutatót dereferálnunk kell a végén.

Ami a kódolási stílust illeti; egyesek egybeírják az `operator` kulcsszót az operátor nevével (`operator=(...)`), míg mások szóközzel választják el a kettőt (`operator =(...)`).

```
class Array
{
    // ...

    // assignment operator
    Array& operator=(const Array& other) {
        if (this != &other) {
            delete[] data;
            capacity = other.capacity;
            data = new int[capacity];

            for (int i = 0; i < capacity; i++) {
                data[i] = other.data[i];
            }
        }

        return *this;
    }
};
```

Hogy markánsabban meglássuk a különbséget a másoló konstruktor és az értékadó operátor között, nézzük meg ezt a példát!

```
Array a01(4); Array a02(5); // we've initialised a01 and a02

Array a03(a01);             // a03 is initialised (copy ctor)
                            // normally: Array a03 = a01;

a01.operator=(a02);         // a01 is assigned to a02
                            // normally: a01 = a02;
```

2.2. Getterek és setterek

A C++ nem támogat speciális szintaxist a getterek, setterek számára (hasonlóan a Javához, de ellentétben a C#-pal), így a hagyományos megoldásokkal tudjuk őket megírni.

```
class Array {
    // ...

    int get_capacity() {
        return capacity;
    }

    void set_capacity(int value) {
        capacity = value;
    }
};
```

A mi esetünkben nincs semmi értelme a kapacitás setterét megírni – sőt, különösen veszélyes ötlet –, csupán a szemléltetés miatt írtam meg.

Foglalkozzunk egy kicsit a getterrel. Vegyük szemügyre az alábbi példát.

```
#include <iostream>

// defintion of class Array

void print_capacity(Array array) {
    std::cout << array.get_capacity() << std::endl;
}

int main() {
    Array array(5);
    print_capacity(array);
    return 0;
}
```

Hibátlanul lefordul; ha kipróbáljuk, megkapjuk, hogy `5`. Viszont tudunk módosítani a `print_capacity` függvényen.

- A paramétert adjuk át referenciaként, ugyanis nyers érték szerinti átadással lemásolódik a teljes objektum a copy konstruktorral – ami költséges.
- A függvény nem módosítja a paraméter semmilyen értékét, úgyhogy adjuk át konstansként.

A módosított függvény tehét így fog kinézni.

```
void print_capacity(const Array& array) {
    std::cout << array.get_capacity() << std::endl;
}
```

Ebben az esetben viszont hibát dob a fordító.

```
pointarray.cpp: In function ‘void print_capacity(const Array&)’:
pointarray.cpp:66:40: error: passing ‘const Array’ as ‘this’ argument
                           discards qualifiers [-fpermissive]
66 |         std::cout << array.get_capacity() << std::endl;
   |                      ~~~~~~~~~~~~~~~~~~~^~
pointarray.cpp:56:13: note:   in call to ‘int Array::get_capacity()’
56 |         int get_capacity()
   |             ^~~~~~~~~~~~
```

A hibaüzenet szerint megsértettük az objektum konstansságát. Konstans referencián nem hajthatunk végre módosításokat, ez világos. A `get_capacity` tagfüggvény nem hajt végre semmilyen módosítát, ezt is jól tudjuk. Akkor mégis miért sértjük meg a konstansságát?

Cseréljük ki a függvényben a gettert a setterre és a probléma máris egyértelműbbé válik. A kinyomtatós parancsoktól egy pillanatra szabaduljunk meg.

```
void print_capacity(const Array& array) {
    array.set_capacity(42);
}
```

```
pointarray.cpp: In function ‘void print_capacity(const Array&)’:
pointarray.cpp:66:40: error: passing ‘const Array’ as ‘this’ argument
                           discards qualifiers [-fpermissive]
66 |         array.set_capacity(42);
   |         ~~~~~~~~~~~~~~~~~~~^~
pointarray.cpp:56:13: note:   in call to ‘void Array::set_capacity(int)’
56 |         void set_capacity(int value)
   |             ^~~~~~~~~~~~
```

A fordító nem tudja eldönteni, hogy mikor módosítja a metódus az objektum belső állapotát, mikor nem. Nincs semmi garancia arra, hogy a getter ne módosítsana valamit a háttérben. Ezt **úgy tudjuk feloldani, ha a getter függvényszignatúrája és -törzse közé beírjuk a `const` kulcsszót**. Így az eredeti függvényünk le fog fordulni.

```
class Array {
    // ...

    int get_capacity() const { // 'const' keyword is inserted
        return capacity;
    }
};
```

2.3. A [] és a () operátor

A `[]` (négyzetes zárójel vagy indexelő) operátort (angolul néha *array subscript operator*) a tömbszerű adatszerkezeteknél használjuk, hogy az i-edik indexű elemét lekérjük. Tartsuk észben, hogy ennek kizárólag egy paramétere lehet (a C-s örökség miatt).

Lássuk be, hogy a `[]`-nak két feladata van, amik ellentmondásosak.

- Írjuk felül az i-edik indexen tárolt adatot. Pl. `data[i] = 42;`
  Ez egy *setter*szerű viselkedés.
- Kérdezzük le az i-edik indexet módosítás nélkül. Pl. `int temp = data[i];`
  Ez egy *getter*szerű viselkedés.

A második esetben garantálnunk kell, hogy a metódus ne módosítsa a tömb belső értékét – hasonlóan a korábbi példához –, ezért jelölnünk kell a végén a `const` kulcsszóval.

```
1  class Array {
2      // ...
3
4      // when we modify the data at the specified index
5      int& operator[](int index) {
6          return data[index];
7      }
8
9      // when we retrieve the data at the specified index
10     const int& operator[](int index) const {
11         return data[index];
12     }
13 }
```

Mivel a C-ben csak egy értéket adhatunk meg a zárójelek között – ellentétben C#-ban, ahol egy mátrix indexelése megoldható így: `matrix[i, j]` –, ezért mátrixok esetén más megoldáshoz kell folyamodnunk. Erre fog megoldást jelenteni a `()` (gömbölyű zárójel) operátor (angolul néha *function call operator*).

```
1  class Matrix
2  {
3      // implementation details
4
5      int& operator()(int i, int j) { return data[i][j]; }
6
7      const int& operator()(int i, int j) const {
8          return data[i][j];
9      }
10 };
11
12 int main() {
13     Matrix m(5, 5);
14     m(1, 2) = 42;
15     return m(1, 2);
16 }
```

Minden típust, amely implementálja a `()` operátort, **funktor**nak (*function operator*, *functor*) nevezünk. Rájuk a következő fejezetben vissza fogunk térni.

2.4. Statikus adattagok és tagfüggvények

C#-ból, Javából már ismerős lehet a `static` kulcsszó. Adattagok és metódusok esetében az osztályszintűséget jelöli, azaz **nem egy specifikus objektumra, példányra, hanem a teljes osztályra vonatkozik**.

Azt is tudjuk, hogy a C++ már rendelkezik egy `static` kulcsszóval, aminek a kontextusai már eleve elég körülményesek voltak. Erre rátesz még egy lapáttal az objektumorientáltsága a nyelvnek, mivel osztályok esetében a megszokott jelentésekkel rendelkezik. *A használatuk* azonban kicsit *eltér* az előbb említett nyelvektől.

Adjunk az `Array` osztályunkhoz egy statikus privát adattagot, ami nyomonköveti, hány példánya él az osztálynak. Mivel privát adattagról van szó, írjunk neki egy nyilvános gettert – és ha már getter, írjuk hozzá a `const` kulcsszót.

```
1  #include <iostream>
2
3  class Array {
4  private:
5      int capacity;
6      int* data;
7      static int count;    // private static member field
8
9  public:
10     Array(int capacity) {
11         if (capacity == 0) capacity  = 1;
12         if (capacity < 0)  capacity *= (-1);
13
14         this->data = new int[capacity];
15         this->capacity = capacity;
16         this->count++;  // increase number of instances
17     }
18     // ...
19
20     // static member function
21     static int get_count() const { return count; }
22 };
23
24 int main() {
25     Array array(5);
26     std::cout << Array::get_count() <<std::endl;
27 }
```

Fordítás során hibaüzenetet kapunk. Azt írja, hogy statikus tagfüggvények nem rendelkezhetnek *cv-minősítőkkel* (cv = `const` és `volatile`).

```
pointarray.cpp:62:32: error: static member function
                             'static int Array::get_count()'
                             cannot have cv-qualifier
62 |         static int get_count() const { return count; }
   |                                ^~~~~
```

A statikus tagfüggvények nem egy konkrét példányra vonatkoznak, hanem a teljes osztályra, ezért nincs értelme a konstansságot nézni – **nyugodtan kivehetjük a kulcsszót**.

Ha így futtatjuk le, akkor egy még csúnyább, még olvashatatlanabb hibaüzenetet kapunk. A sok mangled névből és temporárius fájlnévből kiszűrhetjük, hogy a konstruktor (`in function 'Array::Array(int)'`) egy definiálatlan hivatkozást talál (`undefined reference to 'Array::count'`) a `count` (statikus) adattagra. Ezt még a `get_count` függvénynél is kiírja.

```
/usr/bin/ld: /tmp/ccSIlVLV.o: warning: relocation against '_ZN5Array5countE'
in read-only section '.text._ZN5Array9get_countEv[_ZN5Array9get_countEv]'
/usr/bin/ld: /tmp/ccSIlVLV.o: in function 'Array::Array(int)':
pointarray.cpp:(.text._ZN5ArrayC2Ei[_ZN5ArrayC5Ei]+0x66):
undefined reference to 'Array::count'
/usr/bin/ld: pointarray.cpp:(.text._ZN5ArrayC2Ei[_ZN5ArrayC5Ei]+0x6f):
undefined reference to 'Array::count'
/usr/bin/ld: /tmp/ccSIlVLV.o: in function 'Array::get_count()':
pointarray.cpp:(.text._ZN5Array9get_countEv[_ZN5Array9get_countEv]+0xa):
undefined reference to 'Array::count'
/usr/bin/ld: warning: creating DT_TEXTREL in a PIE
collect2: error: ld returned 1 exit status
```

Itt ismét fel kell idéznünk a változók láthatóságát és élettartamát. Az objektumon belüli adattagok a példányosítás során jönnek létre és az objektum destruálásáig élnek. Azonban a **statikus adattag** jelentése az, hogy osztályszintű, tehát a program teljes életciklusa folyamán láthatónak és elérhetőnek kell lennie. Emiatt úgy kell **az osztályon kívül definiálnunk, mintha egy globális változó lenne**.

```
1  class Array {
2  private:
3      int capacity;
4      int* data;
5      static int count;    // private static member field
6
7  public:  // ...
8      static int get_count() const { return count; }
9  };
10
11 int Array::count = 0;    // definition of private static variable
```

Azért mondom, *mintha* globális lenne, mivel egy tisztán globális változó nem tartozik egy blokkhoz sem. Viszont itt a `private` kulcsszó az osztályra korlátozza a láthatóságot, így egy ilyen kódrészlet fordítási hibát dobna.

```
1  // ...
2
3  int main() {
4      Array array(5);
5
6      // calling a public static member function -> OKAY
7      std::cout << Array::get_count() <<std::endl;
8
9      // referencing a private static member field -> ERROR
10     std::cout << Array::count <<std::endl;
11
12     return 0;
13 }
```

2.5. Típuskonverziós operátorok

Írjunk egy olyan kasztoló operátort az osztályunknak, ami egy hagyományos `int` pointerré konvertálja a tömbünket. A szintaxisa `int* traditional_array = (int*)array;` legyen.

A kasztoló operátorok a következőképp néznek ki: `operator type_to_cast_to() {...}`. **Nem írjuk ki a visszatérési értékét**, a gömbölyű zárójel üres, a `type_to_cast_to` meg állhat több elemből is (mint itt is, a `*` a végén).

A végén akár jelezhetjük is, hogy a tagfüggvény nem módosítja a belső állapotot.

```
class Array
{
// ...

    operator int*() const
    {
        int* traditional = new int[capacity];

        for (int i = 0; i < capacity; i++)
            traditional[i] = data[i];

        return traditional;
    }
};
```

Ugyanezzel az egyszerűséggel implementálhatjuk a `toString`nek megfelelő szöveggé konvertáló operátort. Itt a trükk annyi, hogy szükségünk lesz a C++ `StringBuilder` típusára, amit a `<sstream>` könyvtárból a `std::ostringstream` típus garantál. A `<<` operátor segítségével tudunk konkatenálni. Természetesen a sztringhez szükséges `<string>` könyvtárat include-olnunk kell.

```
#include <string>
#include <sstream>

class Array
{
// ...

    operator std::string() const
    {
        std::ostringstream str;
        str << "{ ";

            for (int i = 0; i < capacity; i++)
            {
                str << data[i];

                if (i != capacity - 1) str << ", ";
            }

            str << " }";

        return str.str();
    }
};
```

2.6. Kiíratás a standard outputra

A típusunkat mostmár képesek vagyunk szöveggé konvertálni, amit felhasználhatunk kiíratásnál.

```
Array array(5);
for (int i = 0; i < array.get_capacity(); i++) array[i] = 42;
std::cout << (std::string)array << std::endl;
// { 42, 42, 42, 42, 42 }
```

Milyen üdvös volna, ha kasztolás nélkül ugyanezt el tudnánk érni! A válasz az, hogy ez megoldható, csupán a `<<` operátort kell implementálnunk.

A `<<` operátor két paramétert vár. Bal oldalán egy `std::ostream&` (output stream) **kimeneti adatfolyam referenciát**[2], a jobb oldalán meg a **kívánt típusú objektum konstans referenciáját**. A kérdés már csak annyi, hogy ki birtokolja a függvényt?

Ha az osztály birtokolja, akkor a kifejezés kiértékelése során nem találja meg az `array`nek a(z) `<<` operátorát, így végeredményül csak az objektum memóriacímét fogjuk megkapni.

```
class Array {
    // ...
    std::ostream& operator<<(std::ostream& out) {
        out << (std::string)(*this);
        return out;
    }
};

int main() {
    Array array(5);
    for (int i = 0; i < array.get_capacity(); i++) array[i] = 42;
    std::cout << (std::string)array << std::endl;
    // 0x55acf5142ef0
}
```

A másik lehetőségünk az az, ha az `std::ostream` osztályában írjuk meg ezt a metódust... Biztosan ezt akarjuk?

Létezik egy harmadik megoldás, mégpedig hogy globális függvényként írjuk meg.

```
std::ostream& operator<<(std::ostream& out, const Array& array) {
    out << (std::string)array;
    return out;
}
int main() {
    Array array(5);
    for (int i = 0; i < array.get_capacity(); i++) array[i] = 42;
    std::cout << array << std::endl; // OK
}
```

Így már helyes eredmény kapunk.

Az figyelhető meg, ha egy operátor két oldalán eltérő típusú objektumok szerepelhetnek, akkor az operátor túlterhelését globális függvénnyel oldjuk meg.

[2] Ebből látható, hogy a `std::cout` típusa `std::ostream`.

2.7. A `friend` kulcsszó

A gyakorlati tapasztalataink szerint a legtöbb adatszerkezet nem támogatja, hogy direkt módon `std::string`gé lehessen konvertálni a példányait. De továbbra is szeretnénk, hogy a szabványos kimenetre ki tudjuk nyomtatni a tartalmát.

Ha megszabadulunk a kasztoló operátorunktól, akkor a globálisan túlterhelt `<<` operátorunk már nem fog működni, ugyanis ettől függ a működése. Megoldhatjuk, hogy ennek a függvénytörzsében történjen a konverzió, viszont ehhez olyan adattagokra van szükségünk, amik privátak, nem tudjuk elérni őket.[3]

```
std::ostream& operator<<(std::ostream& out, const Array& array) {
    std::ostringstream str;
    str << "{ ";

    for (int i = 0; i < array.capacity; i++) { // !
        str << array.data[i]; // !
        if (i != array.capacity - 1) str << ", "; // !
    }

    str << " }";
    out << str.str();
    return out;
}
```

Az olyan szituációt, amikor **egy globális függvénynek egy osztály privát adattagjai számára kell láthatóságot biztosítanunk, a barát (`friend`) függvényekkel tudjuk megoldani**. Az ilyen függvények deklarációit az osztályon belül helyezzük el, a definícióit (jellemzően) kívül írjuk meg.

Az osztályunk az alábbi módon bővül.

```
class Array
{
    // ...
    // cast operator to std::string is deleted

    // declaration of friend function
    friend std::ostream& operator<<(std::ostream& out,
                                    const Array& array);
};

// definition withopud the 'friend' keyword
std::ostream& operator<<(std::ostream& out, const Array& array)
{ ... }
```

[3] Valójában az `Array` osztály esetében pont nincs szükségünk a privát adattagok elérésére, ugyanis a megfelelő getterek, setterek mind nyilvánosak. Ez azonban nem minden esetben adott, így érdemes ismerni a **barát (`friend`) függvények**et

2.8. Öröklődés / Származtatás

Az öröklődéssel már találkoztunk korábbi tanulmányaikból (C#, Java), így a koncepció nem új. A származtatott osztály rendelkezik ugyanazokkal az adattagokkal és tagfüggvényekkel, amit az ősosztálytól... nos, örökölt. A gyerekosztály rendelkezhet ezehfelül még más adattagokkal / metódusokkal, de azokat az ősosztály nem kapja meg.

Vegyük az alábbi példát! Írjunk egy üres `Shape` (síkidom) osztályt (ne feledjük, hogy a fordító generál nekünk 4 tagfüggvényt, így nem teljesen üres az, amit kapunk). Származtassunk le belőle egy `Circle` (kör) és egy `Rectangle` (négyzet) osztályt. Mindketten rendelkezzenek a szükséges adattagokkal, konstruktorrak és legyen területszámító tagfüggvényük (`area`).

```
1  class Shape {};
2
3  class Circle : public Shape // !
4  {
5      double radius;
6
7  public:
8      Circle(int radius = 1)
9      {
10         if (radius < 0) radius *= -1;
11         this->radius = radius;
12     }
13
14     double area() { return 3.14 * radius * radius; }
15 };
16
17 class Rectangle : public Shape // !
18 {
19     double width, height;
20
21 public:
22     Rectangle(double width = 1, double height = 1)
23     {
24         if (width < 0) width *= -1;
25         if (height < 0) height *= -1;
26
27         this->width = width;
28         this->height = height;
29     }
30
31     double area() { return width * height; }
32 };
```

Az eddigi nyelvekkel ellentétben, a C++-ban **származtatásnál ki kell írnunk a `public` kulcsszót a `:` operátor után**. Az alábbi tesztprogrammal próbáljuk ki az osztályainkat

```
1  int main() {
2      Shape shapes[5];
3      return 0;
4  }
```

A programunk egyelőre nem csinál semmi értelmeset, ugyanakkor lefordul – hibamentesnek

tűnik. Ismét elevenítsük fel, hogy a tömb deklarációja esetében is lefut elemenként a default konstruktor (azaz paraméter nélküli, amit a compiler generál(hat)). Módosítsuk a feladatot úgy, hogy adjuk össze az összes síkidom területét!

```
int main() {
    Shape shapes[5];
    double area_sum = 0;

    for (int i = 0; i < 5; i++) {
        area_sum += shapes[i].area();
        // error: no area() member function in class 'Shape'
    }

    return 0;
}
```

Ekkor hibát kapunk, ugyanis a `Shape` osztályunk üres, nem rendelkezik a megfelelő tagfüggvénynel. Adjunk neki egyet, ami visszatért egy alapértelmezett értékkel. Így végeredményül nullát kell kapjunk.

```
#include <iostream>

class Shape {
public:
    double area() { return 0; }
};

// ...

int main() {
    Shape shapes[5];
    double area_sum = 0;

    for (int i = 0; i < 5; i++) {
        area_sum += shapes[i].area();
    }

    std::cout << area_sum << std::endl  // result: 0

    return 0;
}
```

2.8.1. Statikus, dinamikus típus

Most rakjunk a tömbbe eltérő típusú, eltérő méretű síkidomokat. Vajon valóban a síkidomok területeinek összegét kapjuk meg?

```
#include <iostream>

int main() {
    Shape shapes[5];
    shapes[0] = Circle(5);        // 78.54
    shapes[1] = Rectangle(4, 2); // 8
    shapes[2] = Circle();         // 3.14
    shapes[3] = Rectangle();      // 1.0
```

```
9       double area_sum = 0;
10      for (int i = 0; i < 5; i++) {
11          area_sum += shapes[i].area();
12      }
13
14      std::cout << area_sum << std::endl // supposed to be 90.68
15      // but 0 is returned
16
17      return 0;
18  }
```

Az igazság az, hogy nem, 0-át kapunk. És a nyelv szabályai szerint is ezt az eredményt kell kapjuk.

Itt bevezetünk két új fogalmat. Öröklődés esetében egy `T` típusú változónak lehet **statikus és dinamikus típusa**. **Statikus típusnak nevezzük azt, amit fordítási időben ismer meg a compiler. Dinamikus típus pedig az, ami futtatási időben derül ki – ahol ez a típus egy altípusa a `T`-nek**.

Például: a `shapes[0]` változónak a statikus típusa `Shape`, hiszen ez szerepel közvetlenül a kódban, de a dinamikus típusa `Circle`. Amikor végigiterálunk a tömb összes elemén és tfh. nem egyértelmű a fordító számára, hogy melyik tagfüggvényt kell meghívnia, alapértelmezetten a statikus típusának megfelelőt fogja választani.

2.8.2. Object slicing

Ezt úgy tudjuk feloldani, ha az ősosztály megfelelő függvényét virtuálisnak állítjuk be. Ezt a `virtual` kulcsszóval tudjuk elérni. Ennek a párja a származtatott osztályokban az `override`, de ezt C++11-ben vezették be, tehát nem használjuk.

```
1   class Shape {
2   public:
3       virtual double area() { ... }
4   };
5
6   class Circle : public Shape {
7       // ...
8       double area() override { ... } // since C++11
9   };
10
11  class Circle : public Rectangle {
12      // ...
13      double area() { ... } // works fine without it
14  };
```

Sajnos, még így is rossz eredményt kapunk. A probléma a típusok méretében gyökerezik. Egyértelmű, hogy `sizeof(Shape) < sizeof(Circle)` és `sizeof(Shape) < sizeof(Rectangle)`, hiszen a `Shape`-nek nincs egy adattagja sem, a `Circle`-nek és a `Rectangle`-nek meg rendre 1, 2. Tehát a tömb nem foglal le elegendő memóriát arra, hogy az altípusú objektumok beleférjenek.

Ha lennének közös adattagjaik (amit az őstől örökölnének), akkor ezek bekerülnének a tömbbe, de ami túlhalad ezen, az már nem. Ezt a jelenséget nevezzük **object slicing**nak és ez az oka, amiért a területek összege nulla marad.

Például, ha a `Shape` eltárolná a középpontjának koordinátáját (a korábbi `Point` osztály segítségével), akkor a tömb eltárolná az egyes altípusú objektumok saját koordinátáit, de sem a sugárt, sem az oldalakat meghatározó adattagokhoz nem jutna már hozzá.

Ennek egy lehetséges megoldása lehet, ha a tömb nem közvetlenül az objektumokat, hanem a memóriacímüket tárolná el. A mi példánkban gondoskodnunk kell, hogy elkerüljük az inicializálatlan pointereket, ugyanis ezek dereferálása szegmentációs hibához vezet.

```
#include <iostream>

// ...

int main()
{
    Shape* shapes[5];    // array of pointers

    Circle c1(5);        // 78.54
    Circle c2;           // 3.14
    Rectangle r1(4, 2); // 8.0
    Rectangle r2;        // 1.0
    Shape sh;            // 0.0

    shapes[0] = &c1;
    shapes[1] = &c2;
    shapes[2] = &r1;
    shapes[3] = &r2;
    shapes[4] = &sh;

    double area_sum = 0;

    for (int i = 0; i < 5; i++) {
        area_sum += shapes[i]->area(); // !!!
    }

    std::cout << area_sum << std::endl // 90.64 (+ rounding errors)

    return 0;
}
```

2.8.3. Interfészek, absztrakt osztályok

Más objektumorientált programozási nyelvekkel ellentétben, a **C++ nem nyújt nyelvi konstrukciót interfészek és absztrakt osztályok kialakítására**. Azonban hasonló viselkedést ugyanúgy el tudunk érni. Gondoljunk csak bele; a `Shape` osztályunk tulajdonképpen egy interfésznek felel meg a legtöbb nyelvben (feltéve, hogy ignoráljuk az `area` metódus függvénytörzsét).

3. fejezet

Sablonok és az STL

3.1. Sablonok (template-ek)

A sablonok (*template*-ek) segítenek abban, hogy olyan **algoritmusokat és adatszerkezetek írjunk, amik bármilyen típussal működnek**. Közvetlenül a deklarációt megelőző sor előtt meg kell határoznunk a használandó típusparamétereket. Ezeket a `template` kulcsszóval kezdve, `<>` zárójelen belül a `typename` kulcsszóval adjuk meg. Egyszerre több típusparamétert is megadhatunk.

```
template <typename T> // template function
void template_function(const T& var) { ... }

template <typename T, class U> // template class
class TemplateClass
{
    T* data;
    U  variable;
    // ...
};
```

A `typename` kulcsszó helyett használhatjuk a `class` kulcsszót is. A jelentésük megegyezik, pusztán kódstílus és konvenció kérdése, mikor melyiket használjuk.

Amikor felhasználunk egy template-osztályt / algoritmust a kódunkban, akkor a programunkban **példányosul** egy olyan változata, ami az adott típusparaméter szerint jön létre. Függvények esetén a fordító a paraméterekből találja ki a példányosítandó változatot. Ezt a folyamatot **template argumentum dedukció**nak nevezzük (ez osztályoknál nem működik).

```
template <typename T>
void template_function(const T& var) { ... }
template <typename T, class U>
class TemplateClass { ... }
int main() {
    template_function(42);  // instantiation with type 'int'
    TemplateClass<int, double> tc;
    // instantiation with types 'int' and 'double'
    return 0;
}
```

3.1.1. Sablonfüggvények

Nem mindig egyértelmű, hogy mikor melyik típust kell használnia a függvénynek. Ha nem tud dönteni a fordító, fordítási hibát kapunk.

```
template <typename T>
T max(T a, T b) { return a < b ? a : b; }
int main() { return max(1, 1.5); } // 'int' or 'double' ??
```

A megoldás, hogy meghatározzuk expliciten a <> operátorral, hogy milyen típust használjon (tehát nem hagyjuk érvényesülni a sablonargumentum dedukciót).

```
template <typename T>
T max(T a, T b) { return a < b ? a : b; }
int main() {
    double x = max<double>(1, 1.5);
}
```

Figyeljük meg, hogy a függvényünk azt feltételezi, hogy a típusunk rendelkezik relációsjel műveletekkel. Nem okoz-e gondot, hogyha olyan típust adunk meg, ami nem rendelkezik ezekkel?

A válasz az, hogy de, és ez fordítási időben ki is derül – jellemzően valami gusztustalanul hosszú hibaüzenet formájában. A fordító végigmegy a függvény kódján, behelyettesíti a típust, és ha azt tapasztalja, hogy a szóban forgó típusnak nincs definiálva a szükséges operátor, akkor hibát dob. Ugyanez a helyzet, ha olyan adattagra / tagfüggvényre hivatkozunk egy osztály- vagy struktúratípusnál, amije valójában nem létezik.

Ezt nevezzük **duck typing**nak („*If it walks like a duck and it quacks like a duck, then it must be a duck.*”).

```
struct S {}; // has no '<' operator
template <typename T>
T max(T a, T b) { return a < b ? a : b; }

int main() {
    S s01, s02;
    S s = max(s01, s02); // error
}
```

Más nyelvekkel ellentétben (Java, C#), a C++-ban nem határozzuk meg expliciten az ilyen típusra vonatkozó különleges előfeltételeket. A fordítás közben derül ki.

Emiatt belátható, hogy a **sablonok fordítása hosszadalmas**, ugyanis **kétszer kell végigmennie a kódján**; először behelyettesíti a típust, másodszor szemantikailag elemzi az így kapott kódot.

Van lehetőségünk a függvény paraméterezésére is.

```
template <int i>
int add_param(int x) { return x + i; }

int main() {
    int example01 = add_param<5>(1);  // 6
    int example03 = add_param<42>(0); // 42
}
```

Így viszont két példánya lesz a függvénynek a lefordult kódban: lesz egy 5-tel paraméterezett és egy 42-vel paraméterezett változata.

Ha a kódunk tartalmazza egy template paraméterrel rendelkező függvény / osztály definícióját, de **nem használjuk fel sehol sem, akkor nem történik meg a példányosítás**, azaz a bináris futtatható fájlban le se generálódik ennek a kódja.

```
template <typename T> // unused template function
void template_function(const T& var) { ... }
int main() { return 0; } // no instantiation occurred
```

3.1.2. Sablonosztályok

Írjuk meg az `Array` osztályunknak a template-es változatát. Nincs más dolgunk, mint a megfelelő `int`eket kicseréljük `T`-kre.

```
template <typename T>
class Array
{
private:
    T* data; // !
    int capacity;
    static int count;

public:
    Array(int capacity) {
        if (capacity == 0) capacity  = 1;
        if (capacity < 0)  capacity *= (-1);

        this->data = new T[capacity]; // !
        this->capacity = capacity;
        this->count++;
    }

    // replace 'new int[...]' with 'new T[...]'

    T& operator[](int index) { return data[index]; } // !

    const T& operator[](int index) const
    { return data[index]; } // !

    operator T*() const { // !
        T* traditional = new T[capacity]; // !

        for (int i = 0; i < capacity; i++)
            traditional[i] = data[i];

        return traditional;
    }
};

template <typename T> // don't forget about global functions...
std::ostream& operator<<(std::ostream& out, const Array<T>& array)
{
    out << (std::string)array;
    return out;
```

```
}

template <typename T> // ...and static variables
int Array<T>::count = 0; // !
```

A feladat a következő: oldjuk meg azt, hogy ha az osztályt `int` típussal példányosítjuk, akkor működjön teljesen másképp – azaz, csak egyetlen számot tudjunk tárolni benne, rendelkezzen statikus adattaggal, tudjuk a megszokott módon kasztolni, de ne rendelkezzen a `[]` operátorral. Ezt **template specializáció**val tudjuk elérni.

```
template <typename T> // original class
class Array { ... };

// nothing between them

template <> // !!
class Array<int> // template specialisation
{
private:
    int data;
    int capacity;
    static int count;

public:
    Array(int data = 0) {
        this->data = data;
        this->capacity = 1;
        this->count++;
    }

    // converting to 'int' is enough
    operator int() const { return data; }
};

template <typename T>
std::ostream& operator<<(std::ostream& out, const Array<T>& array)
{ // original
    out << (std::string)array;
    return out;
}

std::ostream& operator<<(std::ostream& out,
                         const Array<int>& array)
{ // specialisation
    out << (int)array;
    return out;
}

template <typename T>
int Array<T>::count = 0;     // original
int Array<int>::count = 0;   // specialisation
```

Próbáljuk ki a specializációnak a funkcionalitásait! Azt mondtuk, hogy ne rendelkezzen `[]` operátorral – azonban a getterekről nem említettünk semmit. Mivel a specializáció nem ismeri az „eredeti osztályt”, nem fog rendelkezni getterekkel – ezért hibát kapunk.

```
int main()
{
    Array<int> a01(42);
    Array<int> a02;

    std::cout << a01 << ", " << a02 << std::endl;    // 42, 0

    int size = a01.get_capacity();  // error
    int count = a01.get_count();    // error

    return 0;
}
```

Itt tudatosítanunk kell, hogy a származtatás és a template fogalma nem jár kéz a kézben. Egy template specializáció nem látja az „eredeti osztály” implementációját, ezért ha nem írjuk meg explicite ugyanazokat a műveleteket, nem fog rendelkezni velük.

3.1.3. Dependent scope-ok

Idézzünk fel pár ismeretet korábbról. Létezik a C++-ban a `::` operátor (*scope resolution operator*). Ennek a **bal oldalán** szerepelhet **névtér-azonosító** vagy **osztálynév** (vagy semmi, pl. *globális névtér* esetén). A **jobb oldalán** szerepelhet (többek között) **statikus változó vagy típusnév** (a többi esettel nem foglalkozunk).

```
namespace MyNamespace
{
    static int AAA;

    class MyClass {};
}

class MyClass
{
    static int AAA;
};

int main()
{
    MyNamespace::MyClass object; // [namespace] :: [typename]
    MyNamespace::AAA = 5;        // [namespace] :: [static v.]
    MyClass::AAA     = 42;       // [typename]  :: [static v.]

    return 0;
}
```

Vizsgáljuk meg a következő példát! Az `S` struktúra egy sablonparaméter szerint tartalmazni fog egy statikus `int` típusú változót. Ha ez a paraméter `int`, akkor helyette egy ugyanolyan nevű aliaszt definiált az `int` típusnak.

```
template <typename T>
struct S      { static int X;  }; // static variable declared

template <> // template specialisation
struct S<int> { typedef int X; }; // a new type is defined
```

Élesben így lehetne kipróbálni.

```
int main()
{
    S<double>::X = 42;  // initialisation of static variable
    S<int>::X variable; // equivalent to: int variable;
}
```

Vegyük például az alábbi függvényt, ami az `S` típust használja fel! Egyetlen sorból áll, mégis fordítási hibát kapunk, ugyanis **nem tudja eldönteni a fordító, hogy a `::` után statikus változó, vagy típusdefiníció szerepel**. Úgyhogy **azt feltételezi, hogy statikus változó lesz**. Ezt nevezzük **függő típusozás**nak.

```
template <typename T>
void func()
{
    S<T>::X var_name;   // type or variable?
}
```

Mivel a template-eket kétszer olvassa át a fordító, a második menetben szintaktikai hibát talál (`X␣var_name;`). Annak egyértelműsítésére, hogy valóban típusnévnek lássa, egyszerűen a `typename` kulcsszót kell az elejére helyeznünk.

```
template <typename T>
void func()
{
    typename S<T>::X var_name;  // type or variable?
}
```

Ezzel kiküszöböltük a hibát. Ez a probléma vissza fog térni az iterátoroknál.

3.2. Az STL (Standard Template Library)

A C-ben azt szoktuk meg, hogy ha valmilyen adatszerkezetre szükségünk van, azt magunknak kell megírnunk. A C++ készítői gondoltak erre és létrehozták a **Standard Template Library**t, vagy röviden az **STL**-t. Ebben megtalálhatjuk a legalapvetőbb algoritmusokat és adatszerkezetek, ezzel hatékonyabbá téve a programozásunkat. A tartalmazott elemek mind az `std` névtérben találhatók.

Minden információ elérhető a `https://en.cppreference.com/w/cpp` oldalon. Ebben a jegyzetben csak a legfontosabbakról gyűjtöttem össze a legalapvetőbb információkat, zanzásítva. A vizsgán használhatjuk a dokumentációt, amit természetesen nem kell bemagolnunk, de a legfontosabb fogalmakkal érdemes tisztában lennünk.

3.2.1. Iterátorok

Elemi programozási művelet, hogy végigmegyünk egy adathalmaz összes elemén. Tömbszerű adatszerkezetek esetén végtelenül egyszerű, azonban mások esetében falba ütközünk. Hogyan megyünk végig egy fának a csúcsain? Hogyan férjünk hozzá? Egyáltalán szabad-e?

Erre a megoldásra születtek meg az **iterátorok**. **A belső reprezentációtól függetlenül végig tudunk menni az adathalmazon elemenként**. Egy `std::vector<int>` típusú objektumnak az iterátor típusát így érhetjük el: `std::vector<int>::iterator`.

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

int main() {
    std::vector<int> v = { 1, 2, 3, 4, 5 };

    for (std::vector<int>::iterator i = v.begin();
         i != v.end(); i++)
    {
        std::cout << *i << std::endl;
    }
    return 0;
}
```

A példából az látszik, hogy a viselkedésük kísértetiesen hasonlít a mutatókéhoz. Valóban, hiszen ezek a speciális típusok **rendelkeznek dereferáló operátorral** (`*`) **és inkrementáló operátorral** (`++`). Több operátort is túlterhelhetnek, de alapvetően ez a kettő szükséges, hogy **iterátornak minősüljenek**.

Rendeljük ki egy függvénybe az iterátorra vonatkozó ciklust! A célunk az, hogy *minél általánosabban működjön*, tehát vezessünk be egy template paramétert a konténer számára. Külön nem kell foglalkoznunk azzal, hogy a konténer milyen típusú objektumokat tárol, hiszen az kiderül a fordítás során.

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

template <typename Container>
void iterate(const Container& c) {
    for (Container::iterator i = c.begin();
         i != c.end(); i++)
    {
        std::cout << *i << std::endl;
    }
}

int main() {
    std::vector<int> v = { 1, 2, 3, 4, 5 };
    iterate(v);
    return 0;
}
```

Sajnos hibára fut a programunk. Az előző alfejezetben pont erről a szituációról volt szó, de most látjuk a gyakorlati hasznát. A fordító azt feltételezi, hogy a `Container::iterator` egy statikus adattag, de később jön rá, hogy szintaktikailag helytelen eredményt kap. Mivel nem egyértelmű a hiba forrása, figyelmeztetést is dob a `g++`, hogy hiányzik a `typename` kulcsszó. Írjuk akkor a helyére!

```
template <typename Container>
void iterate(const Container& c) {
    for (typename Container::iterator i = c.begin();
         i != c.end(); i++)
    {
        std::cout << *i << std::endl;
    }
}
```

Azonban így sem fordul le a programunk. Ezúttal a konstansságot sértjük meg. A vektort konstans referenciaként adjuk át, ilyen módon a fordító csak konstans tagfüggvényeket fog engedni, hogy meghívjunk. Azonban *megvan a veszélye, hogy az iterátoron keresztül manipuláljuk a vektor tartalmát, amit nyilván el akarunk kerülni.* Ennek elkerülésére lettek bevezetve a **konstans iterátorok** (`const_iterator`). A konstans iterátorhoz tartozik speciális elejét (`cbegin()`) és végét (`cend()`) meghatározó metódus, de a hagyományosat is használhatjuk.

```
template <typename Container>
void iterate(const Container& c) {
    // c.begin() will work just fine
    for (typename Container::const_iterator i = c.cbegin();
         i != c.cend(); i++) // c.end() is equally ok
    {
        std::cout << *i << std::endl;
    }
}
```

Kezd picit hosszúvá válni az iterátortípus neve. Ezt tudjuk tömöríteni egy **újabb template paraméter** hozzáadásával. Ehhez viszont szükségünk van további függvényparaméterre, hogy helyesen működjön a template paraméter dedukció. Alkalmazzuk azt a bevett szokást, ami az STL algoritmusaira jellemző, hogy **megkapja paraméterül a kezdeti és a végső iterátort** az alábbi módon. Így mostmár semmilyen konstansséggal meg `typename`mel nem kell szenvednünk.

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

template <typename Container, typename Iterator>
void iterate(Iterator begin, Iterator end, const Container& c)
{
    for (Iterator i = begin; i != end; i++) {
        std::cout << *i << std::endl;
    }
}

int main()
{
    std::vector<int> v = { 1, 2, 3, 4, 5 };
    iterate(v.begin(), v.end(), v);
    iterate(v.cbegin(). v.cend(), v); // both work perfectly
    return 0;
}
```

Egyes konténereknél van lehetőség arra is, hogy fordított irányban is bejárjuk – a vektor képes erre is. Az irány megfordításához az `rbegin()` és `rend()` metódusokkal rendelkező `reverse_iterator` típust tudjuk használni. Mivel az algoritmusunkat elég általánosan fogalmaztuk meg, így ezek meghívásával is tökéletesen fog működni – még az `i++`-t sem kell átírnunk! Ennek a változatnak is ugyanúgy létezik a konstans verziója, a `reverse_const_iterator` (`crbegin()` és `crend()` műveletekkel).

```
1  // ...
2
3  int main()
4  {
5      std::vector<int> v = { 1, 2, 3, 4, 5 };
6
7      iterate(v.begin(), v.end(), v);      // iterator
8      iterate(v.cbegin(). v.cend(), v);    // const_iterator
9
10     iterate(v.rbegin(), v.rend(), v);    // reverse_iterator
11     iterate(v.crbegin(), v.crend(), v); // reverse_const_iterator
12
13     return 0;
14 }
```

Iterátor invalidáció: egyes műveletek invalidálhatnak iterátort, mely után szigorúan tilos használni azt.

- Pl. `std::vector`nak a `push_back()` művelete invalidálja az iterátort, ha duplázódik a kapacitás
- De: `std::list`nek a `push_back()` művelete nem invalidálja az iterátort

Iterátorok típusai

1. **Input iterator** vagy **output iterator** – egyirányú, egyetlenszer mehetünk végig rajta. Ha *jobbértéket* olvas be, akkor *input iterator*, ha *balértéket*, akkor *output iterator*.
   Operátorok: `*`, `++`, `==`, `!=`
   Implementálja: `std::istream_iterator`, `std::ostream_iterator`
2. **Forward iterator** – egyirányú, többször mehetünk végig.
   Operátorok: `*`, `++`, `==`, `!=`
   Implementálja: `std::forward_list` (C++11)
3. **Bidirectional iterator** – kétirányú, többször végig mehetünk rajta.
   Operátorok: `*`, `++`, `--`, `==`, `!=`
   Implementálja: `std::list`, `std::set`, `std::map`, ...
4. **Random access iterator** – kétirányú, többször, és akár tetszőleges elemet is kiválaszthatunk.
   Operátorok: `*`, `+`, `-`, `++`, `--`, `+=`, `-=`, `==`, `!=`
   Implementálja: *tömb*, `std::vector`, `std::deque`

3.2.2. Konténerek (adatszerkezetek)

`std::vector`

- könyvtár: `<vector>`
- implementáció a háttérben: **dinamikus tömb** a heapen
- iterátor: **random access iterator**
- specializációk
  - `bool` – 1 bájtnyi tárterületre 8 bit fér, ahol 1 bit jelöl 1 darab logikai változót. Változókat nem tárolhatunk el 1 bájtnál kisebb területen, emiatt a `bool`ok vektora (tömbje) 8-szor hatékonyabb
  - `string` $\Longleftrightarrow$ `std::vector<char>`
- műveletek
  - **konstruktor**
    - megadható a kezdeti méret, de ilyenkor a típusparaméter default konstruktora lefut és beszúrásnál ezen objektumok után fogja behelyezni az elemet
    - default konstruktor, kezdeti méret nélkül
  - `size()` – tárolt elemek száma
  - `capacity()` – teljes kapacitás
  - `[]` operátor – $O(1)$ hatékonyságú, túlindexelésnél nem dob hibát
  - `at()` – olyan, mint az indexelő operátor, de túlindexelés esetén kivételt dob
  - `push_back()`
    - a vektor végébe beteszi az elemet ($O(1)$)
    - ha betelt, megduplázza a tárterületet (ekkor $O(n)$)
    - összességében **amortizált konstans** $O(1)$: minél több elemet szúrunk be, annál ritkábbá válik szükségessé a tömb méretének duplázása, ezáltal a kettő érték annyira kiegyenlíti egymást, hogy konstanssá válik
  - `push_front()` – beszúrás az elejére, $O(n)$, mivel el kell tolnia mindegyiket eggyel
  - `pop_back()`, `pop_front()` – rendre az utolsó / első elemet kiszedi, $O(1)$
  - `reserve()` – üres vektor, kapacitást megváltoztatja, nem inicializál
  - `swap()` – két tömb belső tartalmát felcseréli (egyszerű pointerátállítással)

`std::list`

- könyvtár: `<list>`
- implementáció a háttérben: **kétirányú láncolt lista** (S2L) (*általában*)
- iterátor: **bidirectional iterator**
- műveletek
  - `size()` – tárolt elemek száma
  - `capacity()` – nem létezik
  - `[]` operátor, `at()` metódus – nincsenek, ugyanis $O(n)$ lenne, ezért úgy döntöttek, nem fogják támogatni
  - `push_back()`, `push_front()` – mindkettő $O(1)$
  - `insert()` – $O(1)$, megadjuk a pointert, ami elé beillesztjük
  - `sort()` – sorbarendezi az elemeket (mivel `std::sort()` nem támogatja)

`std::deque`

- könyvtár: `<deque>`
- implementáció a háttérben: **deque adatszerkezet**, röviden: *tömbök vektora*[1] (figyelem: nem mátrix!)
- iterátor: **random access iterator**
- műveletek
  - `size()` – tárolt elemek száma
  - `capacity()` – teljes kapacitás
  - `[]` operátor, `at()` metódus támogatott
  - hasonló metódusok vonatkoznak rá, mint a `std::vector`ra

`std::set`

- könyvtár: `<set>`
- implementáció a háttérben: **bináris keresőfa**, rendezetten tárolja az elemeket
  - előfeltétel, hogy a típus rendelkezzen `<` operátorral, azaz **rendezhető legyen**
- iterátor: **bidirectional iterator**
- műveletek
  - `size()` – halmaz számossága
  - `capacity()`, `[]` operátor – nincsen
  - `insert()` – elem beszúrása, eleve rendezve szúrja be → nincsen `push_back()` és `push_front()` művelete

[1] Bővebben: `https://www.wikiwand.com/en/Double-ended_queue`

`std::map`

- könyvtár: `<map>`
- implementáció a háttérben: **szótár adatszerkezet**, kulcs-érték párokat tárol, általában **piros-fekete fa** áll a hátterében
  - előfeltétel, hogy a kulcstípus **rendezhető legyen**
- iterátor: **bidirectional iterator**, kulcs-érték párral tér vissza
- műveletek
  - `size()` – tárolt elemek száma
  - `capacity()` – nincsen
  - `[]` operátor
    - az operátor a kulcsértéket várja (tehát ha `<std::string,␣int>` párokat tárol, akkor `std::string`et kell megadnunk a `[]` operátorban)
    - adatlekérdezésnél $O(\log n)$ műveletidejű, még elfogadható
    - **ha nem szerepel a halmazban az adott elem, beszúrja azt**
    - nem-konstans tagfüggvényként van definiálva, máskülönben nem tudnánk beszúrni
    - visszatérési értéke: az érték, ami a kulcshoz van rendelve
  - `find()` – adott kulcs-érték pár megkeresése

`std::ifstream`

- könyvtár: `<fstream>`
- implementáció a háttérben: **adatfolyam**
- iterátor: **input iterator**
- műveletek
  - `<<` – adatfolyam fogadása
  - `>>` – adatfolyam küldése
  - `close()` – adatfolyam lezárása (C-ben `fclose()`)

3.2.3. Algoritmusok

Könvytáruk: `<algorithm>`. A legtöbb algoritmus függvényszignatúrája a következő:

```
algo(begin : Iterator, end : Iterator, other_data : AAA) : BBB
```

Tehát iterátorok által meghatározott $[begin, end)$ intervallumon dolgoznak és a legtöbben iterátorral térnek vissza – de természetesen vannak kivételek.

Sokuknak létezik `_if` szuffixummal ellátott változata is, mely egy **funktor**t vár el utolsó paraméteréül. **A funktor egy olyan típus, ami implementálja a `()` operátort**. A legtöbb STL-es algoritmus `bool`lal visszatérő funktort vár el.

`std::copy`

- átmásolja az elemeket egyik adatszerkezetből a másikba (eltérő típusúak is lehetnek)
- a forráskonténer kezdő és végső iterátorát, valamint a célkonténer keldő iterátorát várja el paraméterül

```
#include <algorithm>
#include <vector>

int main() {
    std::vector<int> source = {1, 2, 3, 4, 5};
    std::vector<int> destination;

    std::copy(source.begin(), source.end(),
              destination.begin());

    return 0;
}
```

`std::count`, `std::count_if`

- egy intervallumon megszámolja, hány elem egyezik meg a paraméterül átadott elemnek

```
#include <algorithm>
#include <vector>

int main()
{
    std::vector<int> src = {1, 2, 2, 3, 4, 4, 5};
    int result = std::count(src.begin(), src.end(), item);

    return 0;
}
```

- egy intervallumon megszámolja, hány elemre teljesül az adott funktor

```
#include <algorithm>
#include <vector>

// functor
struct IsEven {
    bool operator()(int item) const {
        return item % 2 == 0;
    }
};

int main()
{
    std::vector<int> src = {1, 2, 2, 3, 4, 4, 5};
    int result = std::count_if(src.begin(),src.end(),IsEven());

    return 0;
}
```

`std::find`, `std::find_if`

- egy intervallumon megkeresi az első elemet, ami kielégíti a feltételt és visszaadja annak az iterátorát
- ha nem talál ilyet, az intervallum végső korlátjával tér vissza – mindig ellenőrizni kell a visszatérési értéket, ugyanis `end()` iterátort tilos dereferálni
- azon adatszerkezeteknél, ahol nem lenne elég hatékony az absztrahált algoritmus, azok előre definiált `find()` metódussal rendelkeznek

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

struct IsEven {
    bool operator()(int item) const {
        return item % 2 == 0;
    }
};

int main() {
    std::vector<int> src = {1, 2, 2, 3, 4, 4, 5};
    std::vector<int>::iterator first_even =
       std::find_if(src.begin(), src.end(), IsEven());
    if (first_even != src.end())
        std::cout << *first_even << std:.endl;
}
```

`std::sort`

- intervallumon sorbarendezi az adatszerkezet elemeit
- muszáj rendelkeznie **random access iterátor**ral, különben nem fog működni a sorbarendezés

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

int main() {
    std::vector<int> src = { 3, 5, 7, 1, 42, 0 };
    std::sort(src.begin(), src.end());
}
```

3.3. Template metaprogramozás

Egy érdekes, előre nem tervezett következménye a C++-os template-eknek a template metaprogramozás. A strukturált vezérlési egységek (ciklusok, stb.) híján a **rekurzió** maradt a fő eszköze, emiatt közel funkcionális programozáshoz hasonló paradigmákkal lehet programozni vele. Nem barátságos, nehéz debugolni, ezért csak érdeklődőknek hívjuk fel rá a figyelmet.

4. fejezet

Függelék

4.1. Különbségek a C és a C++ között

4.1.1. Kulcsszókkal kapcsolatos különbségek

- **A `const` kulcszó**.
  - C-ből örökölt jelentései:
    - `const T var;` – **konstans változó**; az értékét nem lehet módosítani közvetlenül.
    - `const T* var;` – **konstans értékre mutató pointer**; a memóriacímet meg lehet változtatni, az általa mutatott értéket nem.
      - `const T* var` vagy `T const* var` – a kettő ekvivalens, a lényeg, hogy a `const` kulcsszó a `*` előtt szerepeljen.
    - `T* const var;` – **konstans mutató**; a pointer memóriacímét nem változtathatjuk meg, de a rá mutató érték módosítható.
    - `const T* const var;` – mint a memóriacím, mind az általa mutatott érték konstans, nem megváltoztatható.
  - C++-os kontextusai:
    - `const T& var;` – **konstans referencia**; sem a referált változót, sem annak az értékét nem lehet módosítani (bizonyos értelemben olyan, mint a `const T* const var`).
    - `T get_value() const {...}` – **konstans tagfüggvény**; egy „ígéretet” teszünk a fordítónak, hogy a tagfüggvényen belül nem módosítunk semmilyen belső értéket.
- **A `static` kulcszó**.
  - C-ből örökölt jelentései:
    - **statikus lokális változók** – a program teljes életciklusa során a memóriában élnek, de csak abban a **blokkutasításban** használhatók, amelyikben definiáltuk.

```
int increment_var() {
    // executed only at the first function call
    static int var = 0;
    var++;
    return var;
}
int main() { return increment_var(); } // 1
```

– **statikus globális változók** – a program teljes életciklusa során a memóriában élnek, azonban csak abban a **fordítási egységben** használhatók, amelyben definiáltuk. Fordítási időben inicializálódnak.

```
// main.cpp
static int x;
int main() { x = 2; }
```

```
// other.cpp
static int x;
void f() { x = 0; }
```

Akár azonos nevük is lehet, hiszen nem egy fordítási egységben vannak.

– **statikus függvények** – a fordítási egységre nézve lokális függvények.

```
// lib.h
#ifndef LIB_H
#define LIB_H

int f();

#endif // LIB_H
```

```
// lib.cpp
#include "lib.h"

static int g()
{ return 42; }

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

```
// main.cpp
#include "lib.h"
int main() {
    int x = f();
    return g(); // 'g' was not declared in this scope
}
```

– C++-os kontextusai:

– **statikus adattagok** – osztályszintű adattagot jelent.

```
class X { static int n; };
// declaration (uses 'static')
int X::n = 1; // definition (does not use 'static')
```

– **statikus tagfüggvény** – osztályszintű tagfüggvényt jelöl.

```
class X {
    static int n;     // declaration
public:
    static void f() { return n; }
};
int X::n = 7; // definition
```

Megjegyzés: a statikus tagfüggvények nem rendelkezhetnek `const`, `volatile` vagy `virtual` kulcsszókkal.

4.1.2. Függvényekkel kapcsolatos különbségek

- **Implicit függvénydeklaráció**.
  - A C támogatja, alapértelmezetten `int` típussal tér vissza.
  - A C++ ezt nem engedi, fordítási hibát dob.
- **Üres paraméterlistájú függvények** (pl. `int main();`).
  - Jelentése C-ben: bármennyi paramétert át lehet neki adni, de nem fog a függvény vele kezdeni semmit. Figyelmeztetést dob a fordító, ha ilyet teszünk. Nulla paraméteres függvényt az alábbi módon lehet C-ben írni: `int main(void);`. Ezt a megoldást a C++ is támogatja.
  - Jelentése C++-ban: nulla paraméteres függvény. Ha ilyennek adunk át valamilyen paramétert, arra fordítási hibát kapunk.

4.1.3. Típusokkal kapcsolatos különbségek

- **Változó hosszú tömbök** (*variadic length arrays*, VLA).
  - C99-től kezdte támogatni a standard, C11-től viszont opcionálissá vált.
  - A C++ sosem támogatta és aktívan ellenzi ezt a módszert.
- **Sztringliterálok típusa**.
  - C-ben `char[]`.
  - C++-ban `const char[]`.

4.2. Példatár

4.2.1. Az Array osztály

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

class Array {
private:
    int* data;
    int  capacity;
    static int count; // static variable declaration

public:
    // constructor
    Array(int capacity) {
        if (capacity == 0) capacity  = 1;
        if (capacity < 0)  capacity *= (-1);

        this->data = new int[capacity];
        this->capacity = capacity;
        this->count++;
    }

    // destructor
    ~Array() { delete[] this->data; }

    // copy constructor
    Array(const Array& other) : capacity(other.capacity) {
        data = new int[capacity];

        for (int i = 0; i < capacity; i++) {
            data[i] = other.data[i];
        }
    }

    // assignment operator
    Array& operator=(const Array& other) {
        if (this != &other) {
            delete[] data;
            capacity = other.capacity;
            data = new int[capacity];

            for (int i = 0; i < capacity; i++) {
                data[i] = other.data[i];
            }
        }

        return *this;
    }

    // getter
    int get_capacity() const { return capacity; }
    // '[]' operator (setter)
    int& operator[](int index) { return data[index]; }
```

```
53      // '[]' operator (getter)
54      const int& operator[](int index) const { return data[index]; }
55
56      // static member function
57      static int get_count() { return count; }
58
59      // cast operator to int*
60      operator int*() const {
61          int* traditional = new int[capacity];
62
63          for (int i = 0; i < capacity; i++)
64              traditional[i] = data[i];
65
66          return traditional;
67      }
68
69      // friend function declaration
70      friend std::ostream& operator<<(std::ostream& out,
71                                      const Array& array);
72  };
73
74  // static variable definition
75  int Array::count = 0;
76
77  // '<<' operator for printing to 'stdout'
78  std::ostream& operator<<(std::ostream& out, const Array& array)
79  {
80      std::ostringstream str;
81      str << "{ ";
82
83          for (int i = 0; i < array.capacity; i++) {
84              str << array.data[i];
85
86              if (i != array.capacity - 1) str << ", ";
87          }
88
89          str << " }";
90
91      out << str.str();
92      return out;
93  }
```

A teljes *Array* osztály

4.2.2. A Point osztály

```
1   class Point {
2       int x, y;
3
4   public:
5       Point(int x = 0, int y = 0) : x(x), y(y) { }
6                                   // initialiser list
7   };
```

A teljes *Point* osztály

4.2.3. A Shape, Circle és Rectangle osztályok

```
class Shape
{
public:
    virtual double area() { return 0; }
};

class Circle : public Shape
{
    double radius;

public:
    Circle(int radius = 1)
    {
        if (radius < 0) radius *= -1;
        this->radius = radius;
    }

    double area() { return 3.14 * radius * radius; }
};

class Rectangle : public Shape
{
    double width, height;

public:
    Rectangle(double width = 1, double height = 1)
    {
        if (width < 0) width *= -1;
        if (height < 0) height *= -1;

        this->width = width;
        this->height = height;
    }

    double area() { return width * height; }
};
```

A származtatást bemutató *Shape* osztály