Objektum-orientált programozás C++ nyelven

Csapó Ádám Balázs

©2013-2023

Tartalomjegyzék

1. Milyen nyelv a C++? 1

1.1. A C++ és a C közötti főbb különbségek 1
1.2. A C++ által támogatott programozási paradigmák 3
1.3. Típusok fajtái és belső reprezentálásuk 4
1.4. Típusbiztonság 6
1.5. Függvények és függvényhívások 7
1.6. Memóriakezelés a C++-ban 8
1.7. Tömbök és mutatók 9
1.7.1. Tömbök, mint folytonos listák 9
1.7.2. Mutatók 10
1.7.3. `void*` 11
1.7.4. Null mutatók 12
1.7.5. Mutatók és tömbök közötti analógiák, mutató összeadás operátora 12
1.7.6. Karakterláncok, tömbök és pointerek 13
1.7.7. Pointerek összeadása, kivonása; Függvénypointerek 14
1.8. Dinamikus memóriakezelés 14
1.8.1. Memória lefoglalása és felszabadítása a halmon 14
1.8.2. A dinamikus memória mint potenciális veszélyforrás 16
1.9. Kvíz kérdés 17

2. Fordítás és linkelés: hogyan lesz a forráskódból gépi kód? 19

2.1. Deklarációk és definíciók 19
2.1.1. Miből áll egy deklaráció? 20
2.1.2. Scope (hatáskör) 20
2.2. Forrásfájl-típusok és a fordítás menete 21
2.2.1. Előfeldolgozás, fordítás és linkeklés 21
2.2.2. `include` direktíva 21
2.2.3. One-definition rule 22
2.2.4. Linkelés 23

3. Konstansok, referenciák 25

3.1. `const` módosító 25
3.1.1. `const` a pointerek kontextusában 25
3.2. Referenciák 26
3.2.1. A referenciák mögött meghúzódó motiváció 26

3.2.2. Referenciák típusai . . . 27
3.2.3. Bal oldali (*lvalue*) referenciák . . . 27
3.2.4. Konstans referenciák . . . 28
3.2.5. Jobb oldali (*rvalue*) referenciák . . . 28
3.3. Referenciák referenciái . . . 31
3.4. Gyakorlati tanácsok a referenciák használatát illetően . . . 31
3.4.1. Mikor érdemes bal oldali referenciát használni? . . . 31
3.4.2. *Const lvalue* és *lvalue* referenciák közötti distinkció . . . 32
3.4.3. Pointerek és referenciák: Mikor melyiket hasznájuk? . . . 32
3.4.4. Hivatkozások átirányíthatósága . . . 33
3.4.5. Operátor-felültöltés referencián . . . 34
3.4.6. Kollekciók hivatkozásokkal . . . 34
3.4.7. A többi eset . . . 34

4. Structok és osztályok 37
4.1. Osztályok mögötti motiváció, főbb funkciók . . . 37
4.2. Miből áll egy osztály . . . 38
4.3. Osztály deklarációja és definíciója . . . 38
4.4. Adatrejtés – `struct` versus `class` . . . 40
4.5. Konstruktorok: Objektumok inicializálása . . . 41
4.5.1. Alapértelmezett default konstruktor . . . 41
4.5.2. Programozó által definiált konstruktorok . . . 42
4.5.3. Inicializálás versus értékadás . . . 44
4.6. Konstans tagfüggvények és `mutable` . . . 44
4.7. `this` kulcsszó, és referencia visszadása metódusból . . . 47
4.8. `static` kulcsszó függvényeken és osztályokon belül . . . 49

5. Objektumok másolása, értékadása és mozgatása 51
5.1. Dinamikus tagváltozók, mint menedzselendő erőforrások . . . 51
5.2. Dinamikus memória lefoglalása konstruktorral és felszabadítása destruktorral . . . 52
5.3. Copy konstruktor . . . 54
5.3.1. Automatikusan generált copy constructor . . . 54
5.3.2. Saját copy constructor . . . 55
5.4. Copy assignment . . . 56
5.4.1. Automatikusan generált copy assignment operátor . . . 56
5.4.2. Saját copy assignment operátor . . . 57
5.4.3. Copy-and-swap idiom . . . 58
5.5. Rule of 3 . . . 59
5.6. Move constructor és assignmemnt . . . 59
5.6.1. Mit jelent konceptuálisan a mozgatás? . . . 59
5.6.2. Move konstruktor . . . 61
5.6.3. Move assignment . . . 62
5.6.4. Move szemantika és kivételek . . . 63
5.7. A move szemantika és a referencia típusának kapcsolata . . . 63

6. Öröklés **68**
6.1. Specializált típusok 68
6.1.1. A specializált típusok mögötti motiváció 69
6.1.2. Megoldás: publikus származtatás 70
6.1.3. Származtatott típusok memória-szerkezete 70
6.2. Láthatóság változásai örökléskor 72
6.2.1. Publikus öröklés 72
6.2.2. Protected öröklés 73
6.2.3. Privát öröklés 73
6.3. Összefoglaló példa 73

7. Dinamikus polimorfizmus **75**
7.1. Dinamikus polimorfizmus problémafelvetése 75
7.2. Egy álmegoldás: strucc-politika! 76
7.3. Egy másik álmegoldás: típusmezők használata 76
7.4. Jó megoldás: virtuális függvények, absztrakt osztályok 77
7.4.1. Virtuális függvények 77
7.4.2. Virtuális függvények működési mechanizmusa 79
7.4.3. Absztrakt osztályok 80
7.5. Dinamikus polimorfizmus `dynamic_cast` útján 81
7.5.1. A `dynamic_cast` mögötti motiváció 81

8. Standard Template Library alapok **84**
8.1. A Standard Template Library története 84
8.2. Mi az a template? 84
8.2.1. Mire ügyeljünk, amikor template-et használunk? 85
8.2.2. Hogyan készíthetünk template típust? 85
8.3. Az STL szerkezete 87
8.4. Containerek 87
8.4.1. Példa: `std::list` 87
8.4.2. Példa: Fák bejárása `std::deque`-kel 88
8.4.3. Asszociatív konténerek 88
8.5. Iterátorok 89
8.6. Függvények és algoritmusok (`<functional>` és `<algorithm>` header) 90

9. Design patternek a C++ nyelvre adaptálva **92**
9.1. A design patternek fő típusai 92
9.2. Kreációs minták 93
9.2.1. Builder minta 93
9.2.2. Factory minta 93
9.2.3. A Builder és Factory minták összehasonlítása 94
9.2.4. Singleton minta 94
9.3. Strukturális minták 95
9.3.1. Adapter minta 95
9.3.2. Decorator minta 96

9.4. Viselkedési minták 99
9.4.1. Command minta 99
9.4.2. Mediator minta 100
9.4.3. Observer minta 101
9.5. Jellemző kritikák a design patternekkel szemben 103

Irodalomjegyzék **103**

1. fejezet

Milyen nyelv a C++?

Forrásmegjelölés

A fejezetben számos magyarázat és példa Bjarne Stroustrup könyvéből származik: [1].

Bjarne Stroustrup, a C++ dán származású feltalálója, „atyja” szerint „*C++ is a general-purpose programming language providing a direct and efficient model of hardware combined with facilities for defining lightweight abstractions*” (B. Stroustrup, először 1982-ben)

A nyelvet kezdetben (többé-kevésbé) a C nyelv bővítésének szánták, azonban sokat merített a BCPL és a Simula nyelvekből is. Akkoriban a nyelv célja elsősorban a rendszerprogramozás támogatása volt, ugyanakkor lényeges szerepet szánt Stroustrup az absztrakciós lehetőségeknek is: hogy a programozó kényelmesen definiálhasson saját típusokat, és hogy ezeket egymásból származtatva finomítani tudja.

Stroustrupot mindeközben egy fontos nem-funkcionális elv is vezérelte: hogy soha ne kelljen a programozónak a jobb hatékonyság reményében alacsonyabb szintű nyelvhez fordulnia. Ehhez részben hozzá tartozik a „*you shouldn't pay for what you don't use*” elv: magyarán, hogy ha a nyelvnek csak a gyorsabb, alacsonyabb szintű funkcióit használjuk, akkor ne kelljen ‘megfizetnünk’ az egyéb kényelmi funkciók futási költségét.

1.1. A C++ és a C közötti főbb különbségek

A C++ alapegysége az *osztály*, amely az alábbi tulajdonságokat hordozza:

- Adatrejtés (data hiding)
- Biztonságosabb inicializálása a változóknak
- Implicit típus-konverzió, dinamikus tipizálás
- Felhasználó által kontrollálható memória menedzsment

- Operátorok felüldefiniálása (operator overloading)

Ezekről a későbbiekben sok szó fog esni, ha most még nem mindegyik értelme világos, később világossá fog válni.

A C++ több, rugalmasabb lehetőséget nyújt a modularizációra, mint a C. Példaként, a C++-ban már létezik olyan fogalom, hogy névtér (namespace), amelynek segítségével külön scope-ot hozhatunk létre változóknak, függvényeknek és típusoknak. Egy másik példa a modularizációra a felhasználó által definiált típusok (*user-defined types*), azon belül is az osztályok alap-működése. Egy osztályban ugyanis alapesetben minden tagváltozó és tagfüggvény (metódus) privát láthatóságú, ami azt jelenti, hogy csak az adott osztály implementációján belül lehet írni, olvasni, vagy (függvény esetén) meghívni.

A típusellenőrzés is szigorúbb a C++-ban, mint a C-ben, ami egy statikusan típusos nyelvnél hasznos tulajdonság, sok nem várt kellemetlenségtől meg tudja kímélni a fejlesztőt! Például, az alábbi programban egy void típusú pointert próbálunk egész szám típusú pointerként értelmezni (pointerekről is lesz szó később) – ez C-ben elfogadható lenne, de C++-ban explicit konverzió (kasztolás) nélkül nem megengedett:

```
void *ptr;
int* i = ptr; //C-ben OK, C++-ban kasztolni kell, pl.
//int* i = (int *)ptr;
```

További „javítások” a C++-ban a C-hez képest:

- szimbolikus konstansok (#define, de kerüljük, mert globális névteret használ)
- inline-olt függvények
- alapértelmezett függvény-paraméterek
- overload-olt (felültöltött) függvény-nevek
- free store management operátorok (new, delete malloc és free helyett)
- referencia-típus
- template-alapú generikus programozás / template metaprogramozás
- . . . és még sorolhatnánk

Valójában a C++ nyelv napjainkban is olyan gyors ütemben fejlődik (változik), hogy ma már értelme sincs nagyon a különbségeket a fentieken túlmenően felsorolni. A C++ mögötti szabványosító testület – melynek Stroustrup is tagja – az utóbbi években 3-évente újabb és újabb szabványt adott ki. Így jött létre a C++11, C++14, C++17 és C++20. Ezek a szabványok mind lényeges újításokat is tartalmaznak – így egy C++20 szabvány szerint megírt programkód nagyon másképp nézhet ki, mint a 2000-es évi szabvány szerint megírt programkód. Ettől függetlenül a visszafele kompatibilitásról mindig is gondoskodtak: a korábbi szabványokat a mai fordítók is ugyanúgy kezelik.

Ennek folyományaként, mindeközben a C++-ban is megmaradt a C azon képessége, hogy akár hardverszinten kezeljük a gép erőforrásait; így a C++-ból is felhasználhatóak a C könyvtárai, és a C++ nyelv a legtöbb platformon erős támogatottságot élvez.

1.2. A C++ által támogatott programozási paradigmák

A C++ nyelv a fejlesztőknek részére sokféle programozói stílus használatát lehetővé teszi. Ez nehézségeket és lehetőségeket egyaránt felvet: egy adott feladatot nagyon sokféleképpen meg lehet oldani, viszont a programozóra van bízva, hogy hatékony kombinációban alkalmazza a nyelv által támogatott programozási stílusokat.

A modern szoftverfejlesztésben fontos, hogy programjaikban egyszerűen, direktben reprezentálni tudjuk az adott alkalmazási területen felmerülő fogalmakat, úgy, hogy a független fogalmak egymástól függetlenek, az összefüggőek pedig egymással világos kapcsolatban legyenek. Mindezt a praktikum figyelembe vétele mellett célszerű tettünk: öncélúan ne modellezük a domain legapróbb részleteit!

A C++ nyelv sok szempontból biztonságot ad, mivel egy **erősen típusos**, és **statikusan típusos** nyelv. Ez azt jelenti, hogy egyrészt ***minden változónak van egy típusa és csak az adott típusra értelmezett műveleteket lehet rajta végrehajtani***. Másrészt azt is jelenti, hogy a típusok szerinti megfelelő használatot már a fordításkor igyekszik ellenőrizni a C++ fordítóprogram, így típushibákat tartalmazó programot (elvben) futtatni sem tudunk. Az ún. típusbiztonságról és annak limitációjairól a későbbiekben még lesz szó.

Ahogyan korábban említettük, a típusok lehetnek beépített típusok, vagy a programozó által definiált saját típusok is (*structok*, *osztályok*, *enum típusok*). Ezek elősegítik annak a célnak az elérését, hogy az információ lokalizáltan (és így az esetleges bugok is lokalizálhatóan) fejtsék ki hatásukat.

A fenti kereteken belül a C++ alapvetően 4-féle **programozási paradigmát** támogat:

- Procedurális programozás, amely főként a szeriális utasítások, valamint az elágazások és ciklusok alapján történő processzálást, illetve egyszerű adattípusokból felépülő összetett adatstruktúrák használatát helyezi előtérbe (hasonlóan, mint korábban a C, Algol és hasonló procedurális nyelvek)
- Adat-absztrakciókra épülő programozás, amely főként interfészek és ezeket megvalósító (absztrakt és 'valódi') osztályok használatát helyezi előtérbe
- Objektum-orientált programozás, amely a fő hangsúlyt az akár többszintű osztály-hierarchiákra helyezi, dinamikus polimorfizmussal támogatva (ez azt jelenti, hogy a származtatott típusú objektum a szülő' osztály bizonyos függvényeit / metódusait újradefiniálhatja, illetve azokat egyedi funkcionalitással ki is terjesztheti
- Generikus programozás, amely az algoritmusokban előforduló típusokat, illetve konstans értékeket képes parametrizálni, úgy, hogy a fordítóprogram mindig az adott programban használt megfelelő típusok és értékek szerinti kódot generálja le (ezt a dinamikus polimorfizmussal szemben szokás 'parametrikus polimorfizmusnak' is nevezni)

Megjegyezzük, hogy ezeket a stílusokat a C++ nyelv már korai fázisaiban is sokféleképpen támogatta, nevezetesen:

- Osztályok: mind a 4 stílus támogatása user-defined-type-okon (UDT-ken) kersztül
- Publikus/privát osztályváltozók és metódusok: interfész és implementáció megkülönböztetése, ld. adat-absztrakcióra épülő programozás

- Tagfüggvények, konstruktorok, destruktor, user-defined inicializálás (assignment): adat absztrahálásához, objektum-orientált programozáshoz, és egységes nyelv generikus programozáshoz
- Függvény-deklaráció: statikusan ellenőrzött interfészek
- Generikus függvények, parametrizált típusok (később template-ek)

A fő hangsúly azonban minden esetben a stílusok hatékony kombinálásán kell, hogy legyen. Stroustrup írja is, hogy ki nem állhatja, amikor a C++-t „objektum-orientált” nyelvként jellemzik, hiszen sokféle paradigmát támogat (újabban már egyre inkább a funkcionális programozást is)

1.3. Típusok fajtái és belső reprezentálásuk

A típus fogalma magában hordozza az adatok lehetséges értékkészletét, illetve az értékeken elvégezhető műveleteket (operátorokat).

A C++-ban több beépített típus létezik, melyek közül átfedések is léteznek (többféle egész szám típus, többféle előjel nélküli típus, többféle lebegőpontos szám), mivel a nyelv nem szeretne túl sokat feltételezni a futtató számítógép hardveréről.

A főbb beépített típusok a következők:

- Bool típus: `bool` (int-re konvertálva 0 vagy 1 értékű, fordítva 0-ból `false`, minden másból `true`)
- Karakter típusok: `char`, `wchar_t`
- Egész típusok: `int`, `short`, `long`, `long long`
- Lebegőpontos típusok: `float`, `double`, `long double`
- Információ hiánya: `void`
- Ezekből deklarátor operátorokkal más típusokat is létrehozhatunk (mutató, tömb, referencia-típusok, `struct`, `enum`, `enum class`)

Egy adott architektúrán egy adott típus reprezentációjának byte-számát a `sizeof(x)` függvénnyel lehet lekérdezni. **Lényeges azonban, hogy a `sizeof()` függvény csak egy *absztrakt* hosszmértéket ad vissza**, ugyanis a `sizeof()` eredményeként kapott byte-szám valójában nem (feltétlenül) annyiszor 8 darab fizikai bitnek felel meg. Ami minden esetben igaz, az az, hogy **egyrészt egy karakter** (`char` típus) **hossza mindig és mindenkor 1 (absztrakt) byte**. Ennek a többszöröse lehet bármely típus reprezentációs hossza, az alábbi megkötésekkel:

- `1 == sizeof(char)<= sizeof(short)<= sizeof(int)`
- `sizeof(int)<= sizeof(long)<= sizeof(long long)`
- `1 <= sizeof(bool)<= sizeof(long)`

- `sizeof(char)<= sizeof(wchar_t)<= sizeof(long)`
- `sizeof(float)<= sizeof(double)<= sizeof(long double)`
- `sizeof(N)== sizeof(signed N)== sizeof(unsigned N)//N lehet: char, short, int, long vagy long long`

Ahogy említettük, a típusokhoz általában operátorok tartoznak. Példa: aritmetikai operátorok egész típusra:

```
int a = 7; //int tipusu valtozo, melynek neve a
//a valtozot egesz tipusu 7-es ertekre inicializaljuk

a = 9; //ertekadas (nem inicializalas): a erteket 9-re modositjuk

a = a+a; //megduplazzuk a erteket
a += 2; //...majd inkrementaljuk 2-vel
++a; //... majd inkrementaljuk (1-gyel)...
```

Továbbá a memóriában minden bitekből áll, a típus ad a biteknek értelmezést. Például:

- A **01100001** bitsorozat 97-es intnek és ‘a’ karakternek felel meg
- A **01000001** bitsorozat 65-ös intnek és ‘A’ karakternek felel meg
- A **00110000** bitsorozat 48-as intnek és ‘0’ karakternek felel meg

```
char c = 'a';
std::cout << c; //a c nevu karakter erteket irjuk ki, ami 'a'
int i = c;
std::cout << i; //az i nevu int valtozo erteket irjuk ki, ami 97
```

Egy kicsit összetettebb példa:

```
//col es cm kozotti konverzio
int main()
{
  const double cm_per_inch = 2.54;
  int val;
  char unit;
  while(std::cin >> val >> unit)
  {//olvassunk be amig a felhasznalo szeretne
    if(unit=='i'){
      std::cout << val << "in == " << val*cm_per_inch << "cm" << std
         ::endl;
    }
    else if(unit=='c'){
      std::cout << val << "cm == " << val/cm_per_inch << "in" << std
         ::endl;
```

```
    }
    else{
      return 0; //"hibas" mertekegyseg (pl 'q') eseten kilepes
    }
  }
}
```

1.4. Típusbiztonság

A típusbiztonság angolul '*type safety*'. Ez azt jelenti, hogy ideálisan a fordító (statikus típusbiztonság), vagy a futtatókörnyezet (dinamikus típusbiztonság) garantálja, hogy minden változót / adatot csak a típusának megfelelően használunk. Magyarán:

- Változót csak inicializálást követően használunk
- Csak a változó deklarált típusának megfelelő műveleteket végzünk rajta
- A változóra definiált minden művelet érvényes értéket ad a változónak

Az ideális eset a statikus típusbiztonság, hiszen el se szabadna indítani az olyan programot, ahol ez a kritérium felborulhat. A C++ sajnos nem 100 százalékosan statikusan típusbiztonságos, és nem is létezik olyan nyelv, amely 100 százalékban az.

Ezért a második legjobb megoldás a dinamikus típusbiztonság: amikor futás közben érzékeli a futtatási környezet, hogy felrúgtuk a típusbiztonság. A C++ nyelv sajnos ezt sem garantálja 100 százalékosan, de léteznek olyan nyelvek, amelyek igen.

Jó példa a típusosság felrúgására az implicit szűkítés: értékadáskor az új típus nem tudja „teljes felbontásban” az eredeti értéket tárolni:

```
int main()
{
  int a = 20000;
  char c = a;
  int b = c;

  if(a!=b){
    std::cout << "oops! " << a << "!=" << b << std::endl;
  }
  else{
    std::cout << "wow! we have large characters!" << std::endl;
  }
}
```

Egy másik jó példát szolgáltatnak az inicializálatlan változók: a fordító tipikusan szól, de a C++ szabvány nem köti ki, hogy szóljon.

```
int main()
{
  int x; //x "random" kezdoertekre inicializalodik
  char c; //ugyanugy c
  double d; //ugyanugy d, viszont: nem minden bitminta ervenyes
     lebegopontos szam!

  double dd=d; //ezert itt pl. baj lehet (nem masolhatunk invalid leb
     .pontos szamokat)
}
```

1.5. Függvények és függvényhívások

Függvényre már láttunk egy-két példát (main függvény). A függvények a modularizáció egyik alapvető egységét jelentik.

Fontos, hogy a C++ nyelvben a paraméterek **érték szerint kerülnek átadásra, nem identitás (cím) szerint**. (Később látni fogjuk, hogy „kivétel” a referencia, mert az csak egy másik név ugyanarra a memóriatartalomra [1])

Például, az alábbi kódrészletben a függvényen belül eszközölt változások nincsenek kihatással a külvilágra:

```
int f(int a, int b)
{
  a = b+1;
  b = a+2;
  return b;
}

int a = 2;
int b = 3;
c = f(a,b); //a es b erteke nem valtozik, c erteke pedig 6
//pointer eseten is csak a mutatott erteket valtoztathatjuk fv-en
   belul
//magat a cimet nem mert az egy ertek (masolat)... legalabbis az
   eredeti pointerre nem lesz kihatassal
```

Ez a viselkedés nem minden nyelven alap, ugyanakkor ha egyszer megértjük, utána jól kiszámítható viselkedést kapunk.

[1] megjegyzés: a pointer az nem kivétel, mert a pointer egy címet tartalmazó változó, és a címet mint értéket ilyenkor is másolatként adjuk át a függvénynek

1.6. Memóriakezelés a C++-ban

Akármilyen programnyelvvel is ismerkedünk, kiemelten fontos téma a memóriakezelés.

C++-ban a program memóriaterülete az alábbi (konceptuális) partíciókból áll:

- parancssori argumentumok (argc, argv[])
- stack (szabad tár felső része, lefele növekszik)
- heap – avagy halom (szabad tár alsó része, felfele növekszik)
- inicializált adatszegmens
- 0-ra inicializált (inicializalatlan) adatszegmens (BSS)
- kódszegmens (programkód)

Kicsit világosabban:

- Kódszegmens: futtatható kód (hozzáférhető, de nem módosítható dinamikusan)
- Adatszegmens: két része van (inicializált és nem inicializált). Globális és statikus változókat tárol. Ami „nem inicializált” (a programozó által), az a nem inicializált adatszegmensbe kerül, viszont automatikusan 0 értéket kap (a típustól függő null értéket).
- Stack: függvények adatainak (pl. paraméterek, lokális de nem statikus változók, stb.) tárolása (ld. később)
- Heap: dinamikusan allokált adatok tárolása (ld. new és delete operátorok)

Ezen változó típusokat jól szemlélteti az alábbi példa:

```
int initToZero1; // inicializalatlan (a programozo altal) de 0
   erteket kap
static float initToZero2;
int* initToZero3; //mindharom inicializatlan adatszegmens mert
   inicializalatlan globalis vagy statikus valtozok
double initializedD = 20.0; //inicializalt adatszegmens

int main()
{
  int stringLength; //stack, mert lokalis valtozo. fv vegenel eltunik
  static int initToZero4; //inicializalatlan adatszegmens
  static int initialized2 = 10; //inicializalt adatszegmens
  myFunction("something"); //fv-hivas: stack
  int* q = new int[5]; //heap
  delete[] q;
}
```

Kiemelt fontossága miatt külön is nézzük meg a stack-et. Függvényhíváskor ennek tartalma így változik:

- Hívás utáni instrukció címének eltárolása (callback)
- Visszatérés típusának megfelelő méretű hely lefoglalása
- A stack tetejére mutat most az ún. ***stack frame***. Minden ami ezután jön szigorúan a függvényhez tartozik, annak futása után törlődik!
- Függvényargumentumok tárolása. Futás közbn létrehozott változók értékeinek tárolása

Függvény visszatérésekor az alábbi lépések kerülnek végrehajtásra:

- A visszatérés értéke bemásolódik a callback fölötti helyre (megj.: valójában alatta levő, mert a stack lefele növekszik – mégis fordítva képzeljük el)
- Minden törlődik, ami a stack frame pointer fölött helyezkedik el
- Visszatérés értékének pop-olása, értékadás esetén a célváltozóba történő bemásolása
- Callback cím pop-olása, és folytatás

1.7. Tömbök és mutatók

1.7.1. Tömbök, mint folytonos listák

`T` típus esetén `T[size]` jelentése: „size darab T típusú elemből álló tömb”. A tömb indexálása 0-tól (size-1)-ig történik.

```
float v[3]; //v[0], v[1] es v[2]
```

Az esetleges index-túlcsordulás katasztrofális, és sokszor nincs futásidejű ellenőrzés. . . (nézzük meg, hogy Visual Studio beépített fordítója esetében van-e!)

C++-ban egy statikus (stack-en elhelyezkedő) tömb mérete csak konstans (fordításkor ismert) érték lehet. Ha változó méretre van szükségünk, inkább használjunk valamilyen STL-ben megtalálható adatstruktúrát, pl. `std::vector`-t:

```
int myarr[n]; //nem OK
std::vector<int> myvec(n); //OK
```

Ha inicializálólistát használunk, akkor a méret kiszámítása fordításkor automatikus:

```
int myarr[] = {1, 2, 3, 4}; //4-elemu tomb
```

A C++-ban a string-literálok is tömbnek számítanak. A végükön láthatatlan '\0' karaktert tartalmaznak:

- "this is a string"
- sizeof("Bohr")==5
- "Bohr" típusa: const char[5]

C-ben és régi C++-ban (C++11-es szabvány előtt) nem-const típushoz is lehetett string literált társítani. Ez ma már nem lehetséges.

```
char* p = "Platon"; //most mar hiba mert nem const!
char[] p = "Zeno"; p[0] = 'R'; //ha modositani szeretnenk,
    hasznaljunk tombot!

//A stringek tarolasa statikus, ezert visszaadhatoak fv-bol (nem
    tunnek el a stackrol)...
//pl. return "range_error"
```

Megjegyezzük, hogy napjainkban már szinte kivétel nélkül az `std::string` típust érdemes használni, amely általában véve (az ún. Small String Optimizatino-nek nevezett optimalizációtól eltekintve) dinamikus memóriában tárolja a stringeket – így azok helyben módosíthatóak, és hosszukat sem kell előre ismerni.

1.7.2. Mutatók

Világos, hogy objektumokra a nevük alapján mindig hivatkozhatunk (mindaddig, amíg a scope-ban elérhetőek!) Ugyanakkor a C++-ban legtöbb objektum önálló identitással is rendelkezik. Vagyis: két ***lvalue*** (*értékadó operátor bal oldalán álló „bal oldali érték”*) akkor is megkülönböztethető, ha értékük ugyanaz, de más memóriaterületen vannak.

A fentiek okán bármely objektum hozzáférhető, ha két dolgot ismerünk:

- az objektum címét
- az objektum típusát

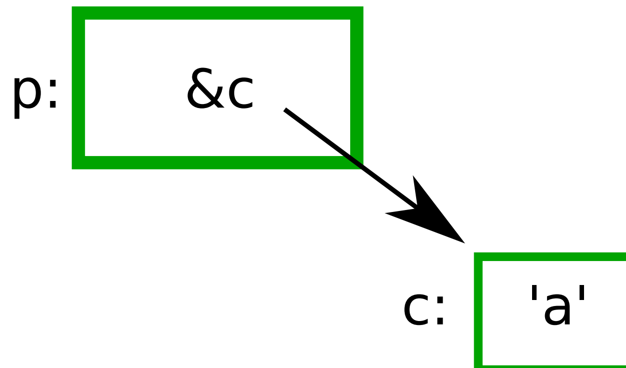
Másra nincs is szükség. A mutató (pointer) objektumok címét jelöli.

```
T* ptr; //T mutatoja tipusu ptr valtozo, amely T tipusu objektum
    memoriacimet tartalmazhatja
```

A mutatókkal kapcsolatos néhány operátor:

- referencia operátor (objektumból cím)
- dereferencia operátor (címből objektum vagy érték)

```
char c = 'a';
char* p = = &c; //p mutato a c cimet tartalmazza, mert & az un.
   address-of operator (referencia operator)
char c2 = *p; //c2 == 'a' ; prefix * operator a dereferencia operator
//rvalue-ra alkalmazva erteket ad vissza. lvalue-ra objektumot:
*p = 'b'; //mostantol c erteke is 'b'
```



A mutatók megvalósítása az ábrán láthatóak szerint direkt módon függ a gép címzési módjától (byte-ot címez? 4 byte-ot egyszerre?). Kevés architektúra tud bitet címezni, ezért a legkisebb „felbontású” pointer char-ra mutat. Ennél kisebb méretű értékeket tárolhatunk:

- bitenkénti logikai operátorok útján
- struct-ok bit-fieldjeiben
- vagy STL bitset objektumokban.

1.7.3. `void*`

Alacsonyszintű kódban előfordulhat, hogy úgy akarunk egy memória címet átadni, hogy nem tudjuk, milyen típus van ott. Ehhez hasznos a `void*` mutatótípus. Egy `void*` típusú változóhoz később bármilyen pointer típust társíthatunk, kivéve függvényre, illetve osztály tagváltozójára mutató pointert.

Két `void*`-ot tesztelhetünk egyenlőségre (mint a pointereknél általában: azonos címre mutatnak-e?), és adott `void*` pointert explicit módon konvertálhatunk más típusra. **Ezt tényleg csak akkor tegyük, ha tudjuk, hogy mit csinálunk!** Ilyenkor végképp a programozó felelőssége, hogy tényleg olyan típus legyen ott, amilyet mond!

Az ilyen típusú pointerre végzett egyéb műveletek nem biztonságosak, mert a fordító nem tudhatja, mire mutat egy `void*`. Felhasználáshoz ezért explicit konverzió kell:

```
void f(int* pi)
{
  void * pv = pi; //OK, implicit konverzio mukodik
  *pv; //hiba, void* nem dereferencialhato!
  ++pv; //hiba, nem inkrementalhatjuk, mert nem ismerjuk a mutatott
     adat meretet

  int* pi2 = static_cast<int*>(pv); //statikus kasztolas rendben
  double* pd1 = pv; //hiba
  double* pd2 = pi; //hiba
  double* pd3 = static_cast<double*>(pv); //nem biztonsagos
  //pv alapbol int-re mutatott, es implementaciotol fugg, mit hogyan
     tarol
  //pl lehet hogy a double-ok 8 byte-os hatarokon kapnak helyet az
     int-ek meg nem
}
```

A `void*` segítségével függvényeknek úgy adhatunk át pointert, hogy semmit sem feltételezhetnek róluk. Alacsonyszintű (hardver) használat esetén ez hasznos, de minden más esetben, amikor ilyet látunk, gyanakodjunk!

1.7.4. Null mutatók

A C++-ban a `nullptr` karaktersor null pointert jelenti: egy pointer, amely nem mutat semmire. Ez az érték bármilyen pointer típushoz társítható, de másfajta beépített típushoz nem:

```
int* pi = nullptr;
double *pd = nullptr;
int id = nullptr; //hiba, i nem mutato
```

Régen gyakran inkább a NULL makrót használták, de annak implementációtól függ a definíciója! A `nullptr` használata ezért tisztább.

1.7.5. Mutatók és tömbök közötti analógiák, mutató összeadás operátora

```
int v[] = {1,2,3,4};
int* p1 = v; //p1 a tomb elso elemere mutat implicit konverzio utjan
int* p2 = &v[0]; //ez is u.az
int* p3 = v+4; //utolso utani elemre mutat
```

Ebben a példában az utolsó sor furcsa, de vannak esetek amikor hasznos (pl. STL iterátorok `end` mutatója az utolsó utáni elemre mutat). Az ilyen mutató értékét sose írjuk vagy olvassuk, csak egyenlőségre teszteljünk vele!

További fontos tanulság a példából: tömb implicit módon mutatóvá konvertálódik. Fordítva ez nem igaz! – hiszen egy pointer alapján nem lehet tudni, hogy milyen hosszú tömb kezdő címe.

Egy adott mutatóra alkalmazott aritmetikai operátor hatása függ attól, hogy milyen típusra mutat a pointer. Ha `T*` típusú pointerhez 1-et hozzáadunk, `p+1` értéke `p + sizeof(T)` lesz!

```
template<typename T>
int byte_diff(T* p, T*q)
{
  return reinterpret_cast<char*>(q) - reinterpret_cast<char*>(p);
  //4-fele explicit kasztolas, szandekosan rondan neznek ki hogy
     kiugorjanak a szemnek
  //a programozo is latja, hogy "itt vigyazni kell"
  //reinterpret_cast: bitmintazatok jelenteset valtoztatja meg
}

void diff_test()
{
  int vi[10];
  short vs[10];

  std::cout << vi << ' ' << &vi[1] << ' ' << &vi[1]-&vi[0] << ' ';
  std::cout << byte_diff(&vi[0], &vi[1]) << std::endl;
}
```

Ha függvénynek tömböt adunk át, implicit módon az első elemre mutató pointerre konvertálódik:

```
int strlen(const char*);

void f()
{
  char v[] = "Annamari";
  int i = strlen(v);
  int j = strlen("Pisti");
}
```

Ha a pointert dereferenciáljuk és módosítjuk a mutatott értéket, az eredeti tömb is módosul (mondhatjuk, hogy kivételesen pass by reference van, de valójában itt is érték szerinti átadás történik, csak egy hivatkozás értékét adjuk át). Persze ilyenkor a függvény nem ismeri a tömb méretét. Ez sok hiba forrása lehet.

1.7.6. Karakterláncok, tömbök és pointerek

A C-stílusú karakterláncok '\0'-val végződnek, ezért hosszuk számítható (pl. az iménti példában) Más esetekben a tömb hosszát is át kell adni a paraméterben:

```
void computeSthg(int* vec_ptr, int vec_size);
```

Ez azonban régies stílus. Ma már inkább konténereket használunk (vector, array, map stb.) amik önmagukban hordozzák az objektumok méretét, sőt, menedzselik annak memóriafelhasználását is.

Végül: ha mindenképpen tömböt adnánk át, nem pointert, akkor tömb-referenciát kell használni, de a tömb hossza is a paraméter része kell, hogy legyen (referenciákról később lesz csak szó):

```
void f(int(&r)[4]);
```

1.7.7. Pointerek összeadása, kivonása; Függvénypointerek

Más objektumokra (tömbökre) mutató pointerek kivonása nem értelmezett eredményt ad. E mellett a pointerek nem adhatók össze (nincs rá értelmes indok se, és a szabvány is kiköti hogy nem lehet)

Ugyanúgy azonban, ahogy az objektumoknak van címük, a függvénytörzshöz generált kódnak is van címe a kódszegmensben. Pointerek ezért függvényekre is definiálhatók, azonban a szintaxisnál oda kell figyelni a megfelelő zárójelezésre:

```
int* fp(char*) //fp fuggveny int-re mutato pointert ad vissza
int (*fp)(char*) //int-et visszaado fv-re mutato pointer
```

Ezen kívül a kódot nem módosíthatjuk derferencia-operátorral (na vajon miért?). Ezért függvénypointer csak fv meghívására és címének másolására használható.

```
void error(string s){/*...*/}
void (*efct)(string); //pointer olyan fv-re, mely stringet var es
   semmit sem ad vissza

void f(){
  efct = &error;
  efct("error"); //fv-hivashoz nincs szukseg dereferencia-operatorra,
      automatikus!
}
```

Itt nem kötelező egyébként az error függvény címét sem használni, implicit konverzió miatt.

1.8. Dinamikus memóriakezelés

A C++-ban minden névvel ellátott objektum a scope-jának megfelelő élettartamú. Sokszor hasznos azonban, ha scope-tól független élettartalmú objektumot hozunk létre – például amikor szeretnénk, hogy egy függvény visszatérése után a változó továbbra is használható legyen, még akkor is, ha a függvényen belül hoztuk létre

1.8.1. Memória lefoglalása és felszabadítása a halmon

A `new` operátor éppen erre való. Az operátor segítségével lefoglalt memória-területet felszabadítani a `delete` operátorral lehet.

A `new` operátorral létrehozott objektumok a halomban (heap) helyezkednek el (avagy free store). Meghívásakor egy pointert kapunk vissza arra a memóriaterületre, amelyet lefoglaltunk. A pointer

típusa pedig attól függ, hogy a new operátor után milyen lefoglalandó típust adtunk meg. Mindez működik alaptípusokra, osztályokra és tömbökre is. Például:

```
int* pi = new int;
//...
delete pi;

vector<int>* pvi = new vector<int>(n);
//...
delete pvi;

char* sp = new char[5];
//...
delete[] sp;
```

Ahhoz, hogy a new és a delete működjön, a futtatási környezetnek találnia kell (az esetlegesen már igencsak particionált) heap-en egy a változó méretének megfelelő összefüggő területet, és ennek lefoglalását el is kell könyvelnie annak érdekében, hogy később a további változóknak is le tudjon foglalni területeket (illetve hogy a területet fel is tudja szabadítani, ha például egy tömbről van szó). Ezért az így létrehozott változók, objektumok kicsit nagyobbak (is lehetnek), mint a stack-en vagy adatszegmensben létrehozottak. Legtöbbször ez nem jelentős overhead, de sok kis objektum esetén korlátot szabhat a dinamikus memóriahasználatnak.

Ha a new operátor nem tud több területet foglalni, `bad_alloc` típusú kivételt dob:

```
void f()
{
  vector<char* >v;
  try{
    for(;;){
      char* p = new char[10000];
      v.push_back(p); //ugyelunk arra hogy maradjon referenciank a
         mem.teruletre
      p[0] = 'x';
    }
  }
  catch(std::bad_alloc){
    std::cerr << ``Memory exhausted!'' << std::endl;
  }
}
```

Nem árt vigyázni: ha van virtuális memória is, akkor a merevlemezre kezd el írogatni a program, és csak akkor dob kivételt, amikor az is betelik

1.8.2. A dinamikus memória mint potenciális veszélyforrás

A halomra mutató pointerekkel rengeteg gond van: a programozók számára – ahogy a gyakorlat is mutatja – gyakran nehéz a `new` operátorral lefoglalt memóriát észben tartani és felszabadítani. Több hibalehetőség is van:

- `new` használatát követően a `delete`-ről elfelejtkezünk (memória-szivárgás).
- A `new`-t követően kivételt dob a program, így sohasem szabadul fel a memória
- korai `delete`: meghívjuk a `delete`-et, de még létezik arra a memória-rekeszre mutató másik pointer is, amit később használunk, miközben a futtatási környezet már esetleg más változónak odaadta azt
- kétszeres `delete`: rossz, ha nem idempotens egy objektum destruktora, ha csak időközben már más változó számára lefoglalta a rendszer az adott memória-rekeszt.

```
int* p1 = new int{99};
int* p2 = p1; //ajjaj...
delete p1; //p2 ezek utan nem mutat konzisztens allapotra!
p1 = nullptr; //hamis biztonsagerzetet ad
char* p3 = new char{'x'}; //itt akar felul is irhattuk az eredeti 99-
   et!
*p2 = 999; //itt lehet hogy ezert pont a karaktert irjuk felul
std::cout << *p3 << std::endl; //nem biztos, hogy 'x'-et kapunk
```

Tekintsük meg ezt a példát is:

```
void sloppy()
{
  int* p = new int[1000];
  //...
  delete[] p;

  //... kicsit varunk

  delete[] p; //de sloppy() mar nem is birtokolja *p-t!
}
```

Ezekre a kihívásokra számos megoldás kínálkozik, azonban mindegyik egyfajta fegyelmezettséget igényel a programozó részéről:

- Használjunk ún. manage-elt objektumot, amely handle-t ad a területre és garantáltan törli a scope megszűnésekor. Példák:
  - string, vector és egyéb STL konténerek
  - unique_ptr, shared_ptr (C++11 óta)

  - külső kvázi-sztenderd könyvtárak: boost shared ptr

- Használjunk inkább referenciát.

A fentiek közül az első módszert RAII technikának is nevezik (Resource Allocation is Initialization). Noha osztályokról még nem esett szó, íme egy példa, amely másodszori olvasásra mindenképpen érthető kell, hogy legyen:

```
class someResource
{//belso reprezentaciok, ptr-ekkel, stb.
public:
  someResource(){//...eroforrasok lefoglalasa}
  ~someResource(){//... eroforrasok felszabaditasa}
};
```

Az STL vector osztály éppen ilyen elven működik! A *push_back()* hívások például a halmot érintik:

```
void f(const string& s)
{
  vector<char> v;
  for(auto c: s)
    v.push_back(c);
}
```

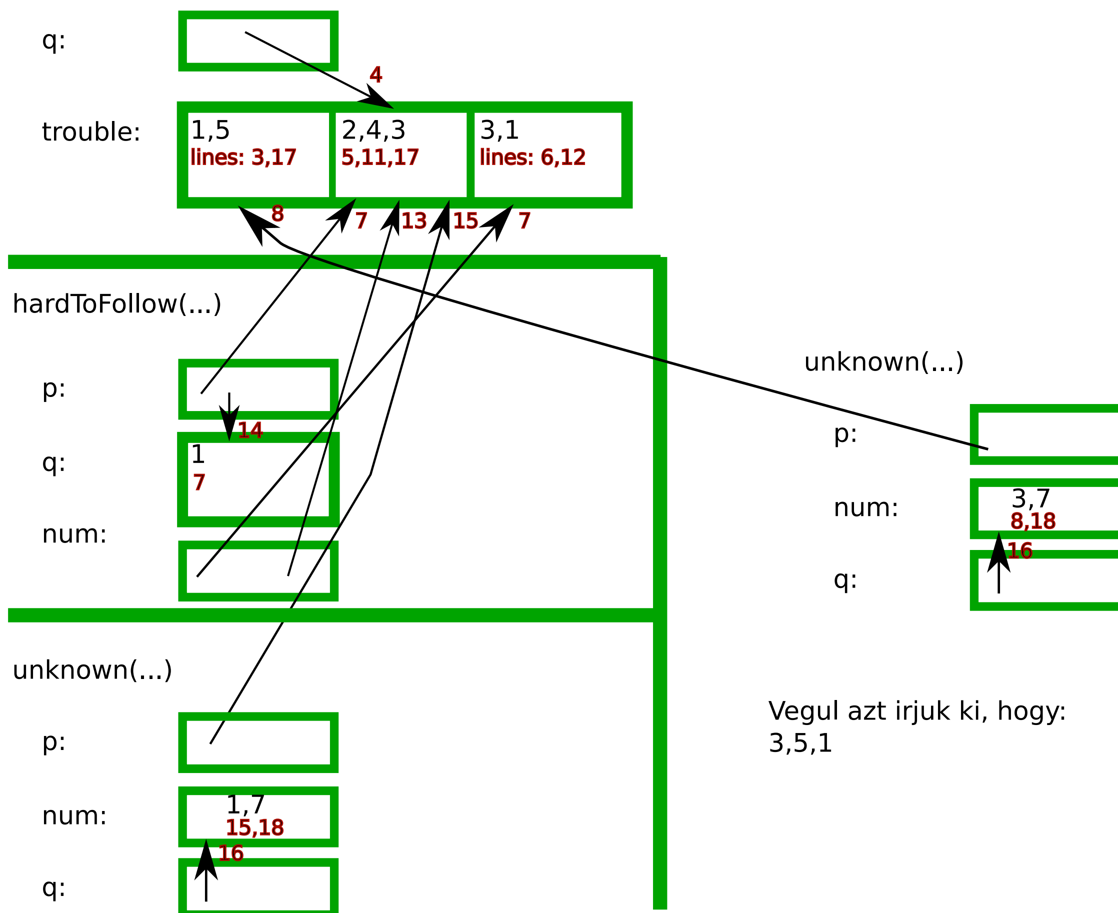
1.9. Kvíz kérdés

Mit csinál az alábbi kód?

```
void unknown(int* p, int num)
{
  int* q = &num; //line 16
  *p = *q + 2; //line 17
  num = 7; //line 18
}

void hardToFollow(int* p, int q, int *num)
{
  *p = q + *num; //line 11
  *num = q; //line 12
  num = p; //line 13
  p = &q; //line 14
  unknown(num, *p); //line 15
}

main()
{
  int* q; //line 1
```

```
    int trouble[3]; //line 2
    trouble[0] = 1; //line 3
    q = &trouble[1]; //line4
    *q = 2; //line 5
    trouble[2] = 3; //line 6
    hardToFollow(q, trouble[0], &trouble[2]); //line 7
    unknown(&trouble[0], *q); //line 8

    std::cout << *q << " " << trouble[0]; //line 9
    std::cout << ", " << trouble[2] << std::endl; //line 10
}
```

A megoldás itt látható:



2. fejezet

Fordítás és linkelés: hogyan lesz a forráskódból gépi kód?

Forrásmegjelölés

A fejezetben számos magyarázat és példa Bjarne Stroustrup könyvéből származik: [1].

2.1. Deklarációk és definíciók

Mielőtt egy nevet, azonosítót használhatunk egy C++ programban, azt a nevet típusa megjelölésével együtt deklarálnunk kell. A **deklaráció** nagyvonalakban azt jelenti, hogy a változó típusát és nevét meg kell adnunk, hogy a fordító tudja, hogy az adott név milyen fajta entitásra fog vonatkozni.

```
char ch;
string s;
auto count = 1;
const double pi = 3.1415926535897;
extern int error_number; //(csak ez meg a kovetkezo nem definicio is
   egyben! ld. lentebb)
double sqrt(double);

const char* name = "Njal";
const char* season[] = {"spring", "summer", "fall", "winter"};
vector<string> people{nae, "Skarphedin", "Gunnar"};
```

Szemmel láthatóan egy deklaráció többet is megtehet, mint hogy névhez típust rendel. Legtöbb fenti példa egyben definíció is (nemcsak deklaráció).

A **definíció** ugyanis olyan deklaráció, amely már minden információt tartalmaz ahhoz, hogy a programban az adott entitást használni is tudjuk.

Konkrétabban: ha valaminek a reprezentálásához *memóriára* van szükség, akkor azt a memóriát a definíció le is foglalja. Ezt a distinkciót felfoghatjuk úgy is, mint az interfész és implementáció közötti különbséget: a deklaráció lenne az interfész, míg a definíció az implementáció.

A C++ ezt a két fogalmat olyannyira megkülönbözteti, hogy a konvenció és a jó gyakorlat szerint a deklarációkat (interfész) és definíciókat (implementáció) külön típusú fájlokban határozzuk meg: előbbiek az ún. *header file*-okban, míg utóbbiak a *forrásfájlokban* jelennek meg.

A fenti példára visszatérve: a példában szinte mindegyik sor memóriafoglalással is jár, kivéve az `error_number` változót (az `extern` minősítő miatt, ami éppen azt jelenti, hogy „*ez nem definíció!*") és a `sqrt()` függvényt.

2.1.1. Miből áll egy deklaráció?

Túl sok leegyszerűsítés nélkül deklaráció (változók és függvények esetén például) sorrendben az alábbi 5 részből áll:

- Opcionálisan: prefix minősítők (pl. `static` vagy `virtual`)
- Egy alaptípus (pl. `vector<double>` vagy `const int`)
- Egy deklarátor (pl. `p[7]` vagy `n`)
- Opcionálisan: szuffix minősítők (pl. `const` vagy `noexcept`)
- Opcionálisan: inicializáló vagy függvénytörzs (pl. `={7,5,3}` vagy `{return x;}`)

A függvény- és névtér-definíciókat leszámítva mindig kötelező pontosvesszővel lezárni a deklarációt.

2.1.2. Scope (hatáskör)

Minden deklaráció valamilyen scope-ba (hatáskör) bevezet egy új nevet. A scope lehet:

- Lokális: a nevet függvényben vagy lambda kifejezésben deklaráltuk. A deklaráció pontjától az adott blokk végéig tart a hatásköre (egy blokkot egy -pár zár közre)
- Osztály: egy név „tagnév" vagy „osztály tagnév" ha egy osztályon belül, de annak bármely függvényén, lambdáján, osztályán vagy enum osztályán kívül definiáltuk
- Névtér: névtérben, de függvényen, lambdán, osztályon, enum osztályon kívül definiált név. Ilyenkor a névteret lezáró kapcsos zárójelig él a név.

A scope lehet továbbá:

- Globális: bármely függvényen, osztályon, enum osztályon vagy névteren kívül deklarált név. Ekkor a deklaráció pontjától az adott fájl végéig él a név. Más fordítási egységekből is el lehet az ilyen változókat érni (ld. később az `extern` minősítőt)

- Utasítás: for, while, if vagy switch utasítás ()-zárójelein belül definiált név az utasítás végéig él
- Függvény: goto-val hivatkozott label (soha ne használjuk!!) az adott függvény törzséhez tartozik

2.2. Forrásfájl-típusok és a fordítás menete

A C++-ban a forrásfájlokból a futtatható állomány (vagy statikus / dinamikus könyvtár) előállítása 3 lépésben történik: előfeldolgozó futtatása, fordítási egységek object fájlra történő lefordítása, majd linkelés.

2.2.1. Előfeldolgozás, fordítás és linkeklés

C++ nyelven a forráskódokat a fordítóprogram az alábbi lépésekben dolgozza fel:

- Preprocesszálás (makrók, #include-dal történő hivatkozások betöltése) → eredmény: translation unit (fordítási egység, egy ilyen általában egy .cpp fájlból keletkezik)
- Translation unit feldolgozása a C++ nyelv szabályai szerint

A külön fordíthatóság érdekében a programozónak minden forrásfájlban minden információt meg kell adnia ahhoz, hogy a fordító izoláltan elemezni tudja a translation unit-okat (jelesül: minden változót, függvényt stb. legalábbis deklarálnia kell) Látni fogjuk, hogy mindez az ún. header file-ok segítségével könnyebb. Egy header file-t akár több .cpp fájl is „behivatkozhat”, így elkerülhető, hogy minden .cpp fájlban mindent újra deklarálni kelljen csak azért, mert külön fordítási egység.

Linkelés

A linkeklés célja, hogy a külön lefordított fordítási egységek eredményéül kapott object fájlokat egyetlen futtatható állományba (vagy statikus / dinamikus könyvtárba) egyesítsük.

Látni fogjuk, hogy mivel a program teljes egészében valójában a linkelés eredményeképpen „ér össze”, ezért bizonyos típusú hibákat csak a linker látható. Ilyen például, amikor egyazon nevű változót két fordítási egységben is definiáltunk (ez az ún. *one-definition rule* ellen megy!), vagy pedig eltérő típussal definiáltunk. Erről a későbbiekben még lesz szó.

2.2.2. `include` direktíva

A *header file*-ok (melyek kiterjesztése tipikusan `.h` vagy `.hpp`) forrásfájlokban történő „behivatkozása” az ún. `include` direktíva segítségével történik:

```
#include <iostream>
#include "sajat_konyvtar.h"
```

Az *include* direktíva annyit mond a preprocesszornak, hogy másolja be a header file tartalmát a fordítási egység azon pontjára, ahol a direktíva szerepel.

Általában jó stílus, ha header file-ban csak interfészt adunk meg, nem definiálunk dolgokat (kivéve amit esetleg inline-olni szeretnénk, ld. majd később a 4.3. szakaszban). Így kisebbek lesznek a lefordított fájok, melyek az include-olt header file-ok szó szerinti másolatát tartalmazzák!

Másrészt egyazon header file-t több forrás is include-olhatja. Ezek után ha az implementáció az adott header file-ban változna, az összes include-oló forrást újra le kellene fordítani! Ha viszont az implementáció egyetlen külön .cpp fájlban lakik, akkor csak azt kell újra lefordítani (illetve linkelni a többivel)

Ehhez kapcsolódóan megjegyezzük, hogy a one-definition rule miatt is lényeges lehet, hogy a több fordítási egységből hivatkozott header file ne tartalmazzon definíciókat, hiszen linker a külön fordítási egységeket külön-külön fogja megkapni.

A fentiek alapján így használhatunk pl. header file-okat:

```
//s.h:
struct S{int a; char b;};
void f(S*);

//file1.cpp
#include "s.h"
//itt hasznalhatjuk f() fv-t

//file2.cpp
#include "s.h"
void f(S* p)
{
  //itt pedig definialjuk
  //ha az implementacio valtozik, csak file2.cpp-t kell ujra
     forditani!
  //az s.h fajlt include-olo tovabbi forrasokat viszont nem kell
}
```

2.2.3. One-definition rule

A one-definition rule szerint minden fordítási egységben, és az egész programban is csak egyszer lehet adott nevű entitást definiálni. Ezért fel kell készülnünk arra az esetre, ha mégis van olyan header file amiben definíció is szerepel. Az ilyen header file-t közvetetten sem include-olhatjuk többször!

Például gondoljuk meg, mi történne, ha include-olnánk két header file-t, és mindkettő include-olná egyazon, harmadik header file-t is, melyben történetesen definíció is szerepel?

Az egyes fordítási egységek szintjén az ilyen helyzetek ellen védenek az ún. *header guard*-ok, melyeket szintén az előfeldolgozó dolgozza fel.

Ezek használata esetén a header file két részből áll:

- 1. *header guard*: védelem az ellen, hogy adott header file-t egyazon fordítási egység többször include-olja
- 2. Maga a tartalom

Például:

```
//ez a ket sor a header guard. ADD_H barmilyen egyedi nev lehet.
   Altalaban a h fajl nevet hasznaljuk.
#ifndef ADD_H
#define ADD_H

int add(int x, int y); //fuggveny deklaracio

//header guard vege
#endif
```

Ez azt jelenti, hogy amikor az adott fordítási egységbe már másodszor hivatkoznánk be az adott header file-t, akkor az `ADD_H` változó már definiálva lenne, így az `ifndef` (if not defined) igaz ága kétszer nem kerülne bemásolásra.

A fentiek alapján a fejlesztéseink során jó gyakorlat lehet, ha egyetlen header file-t hozunk létre az összes deklarációval, melyet a deklarált interfészeket implementáló `.cpp` file-ok hivatkoznának be. Ezt a header file-t később inkrementálisan felbonthatjuk, ahogy egyre összetettebbé és következésképpen modularizáltabbá válik az alkalmazás.

Fontos, hogy a header file-okban előforduló változók neveit (ha egyáltalán van ilyen, mert ritkán érdemes ezt az utat követni!) az `extern` kulcsszóval deklaráljuk, így elkerülve azt, hogy véletlenül memória-foglalás is történjen. Ezt követően a memória-foglalás / értékadás megtörténhet a megfelelő `.cpp` fájlban.

2.2.4. Linkelés

A programozó felelőssége, hogy minden fordítási egységben, translation unit-ban deklarálva legyenek azok a névterek, osztályok, függvények, stb. amiket használ, és hogy a különböző translation unit-ok ezeket konzisztens módon használják.

Például az alábbi kódrészlet e tekintetben helyes:

```
//file1.cpp
int x = 1;
int f() { /* valamit csinal */}

//file2.cpp
extern int x;
int f();
void g(){ x = f(); }
```

Ebben a példában viszont 3 hiba is van:

```
//file1.cpp
int x = 1;
int b = 1;
extern int c;

//file2.cpp
int x; //azt jelenti, hogy x=0
extern double b;
extern int c;
```

Ugyanis:

- x kétszer van deklarálva és definiálva
- b-t kétszer deklaráltuk eltérő típussal
- c nincs definiálva

Lényeges, ahogy korábban említettük, hogy a fordító ezeket a hibákat nem látja, mivel csak külön-külön elemzi a fordítási egységeket. Ezeket a hibákat tehát a linker fogja észrevenni és jelezni. A linker hibaüzenetei azonban sokszor nehezebben értelmezhetőek (különösen kezdő programozók számára), mivel nem egy konkrét kódsorra hivatkoznak (már a köztes, *object file*-okat dolgozza fel!).

Visszatérve, amikor azt írjuk, hogy `#include<iostream>`, majd felhasználjuk az `std::cout` függvényt, akkor azt tudjuk, hogy:

- Az iostream header deklarálja az `std` névtérben levő `cout` függvényt
- A C++ runtime support könyvtár implementálja is, ehhez pedig a fordító automatikusan linkeli a kódunkat.
- Ha nem létezne `iostream` header file, akkor az `std::cout` felhasználása előtt az összes szükséges deklarációt meg kellett volna adnunk, ugyanis adott fordítási egységben nem használható fel olyan név, ami nincs deklarálva!

3. fejezet

Konstansok, referenciák

Forrásmegjelölés

A fejezetben számos magyarázat és példa Bjarne Stroustrup könyvéből származik: [1].

3.1. `const` módosító

C++ nyelven a `const` módosító használatakor azt mondjuk a fordítónak, hogy: „*megígérem, hogy ezt az értéket nem változtatom meg*”

Ez a módosító változók deklarálásakor / definiálásakor, illetve függvények szignatúrájának (interfészének) deklarálásakor használatos.

Egy `const` változó (mint `const int`, vagy `const std::string`) esetén nem kell attól tartani, hogy a változó értékét bármelyik más kódrészlet bármikor módosítaná. Éppen ezért a `const` változókat kötelező inicializálni – deklarálásukkal egyidőben definiálni is kell őket, ellenkező esetben a fordító hibát jelez.

Interfészek meghatározásánál a `const` eredményeképpen nem kell attól tartani, hogy a függvénynek átadott adat a függvényen belül módosulni fog. Értelemszerűen ez csak pointerek illetve (ld. később) referenciák esetében tud érdekes lenni, mivel minden más argumentum érték szerint kerül átadásra (a függvény törzse egy „másik” változót lát, mint a függvény meghívásának kontextusában elérhető változó).

3.1.1. `const` a pointerek kontextusában

Mutatók esetében nem mindegy, hogy mire értjük a „konstans” szót: a mutató mint érték konstans (a változó később más címre már nem mutathat), vagy a mutató által hivatkozott címen levő adat értéke konstans (a változó később mutathat más címre, de rajta keresztül a címen levő érték nem módosítható).

```
char *const cp; //konstans mutato, ami karakterre mutat (constant
    pointer)
char const* pc; //mutato, ami konstans karakterre mutat (pointer to
    const)
const char* pc2; //mutato, ami konstans karakterre mutat (pointer to
    const)
const char* const cpc; //konstans mutato, ami konstans karakterre
    mutat
```

Ezen esetek megkülönböztetésére sokan használják a „hátulról olvasás” módszerét:

- „cp egy konstans pointer karakterre”
- „pc egy pointer konstans karakterre”
- „pc2 egy pointer karakterre, amely konstans” (ami ugyanazt jelenti, mint az előző eset)

Végső soron tehát a fő különbség, hogy a `const` módosító a csillagtól balra (az érték konstans) vagy a csillagtól jobbra (a pointer mint cím konstans) helyezkedik el. Ha a csillag mindkét oldalán megtalálható a `const` módosító, akkor maga a cím is, és a címen levő érték is (az adott pointeren keresztül nézve!) konstans lesz.

Konstansra mutató pointernek adható olyan cím, amely nem konstans változót tartalmaz. Ebből baj nem származhat, legfeljebb a fordító nem engedi, hogy a pointeren keresztül, azt dereferenciálva módosítsuk a változót...

Konstans változó címét azonban nem adhatjuk át nem konstansra mutató pointernek, mert ezáltal a mutatott változó értéke mégis valamilyen módon módosítható lenne (dereferencia operátor útján)!

```
int a = 1;
const int c = 2;
const int* p1 = &c; //OK
const int* p2 = &a; //OK
int* p3 = &c; //hiba: int* valtozo inicializalasa const int*-el
*p3 = 7; //ez gond lenne, pont ezert hiba az elozo sor
```

3.2. Referenciák

A referenciák sok esetben megkönnyítik a más objektumokra való hivatkozások létrehozását, átadását, mivel memória-címek helyett absztraktabb szinten (nevekkel) operálnak, és mivel élettartamuk alatt kötelezően egy és csakis egy változóra tudnak hivatkozni.

3.2.1. A referenciák mögött meghúzódó motiváció

A pointerek használata többféle kihívás elé állítja a programozót:

- A pointerek használatakor eltérő szintaxist használunk eltérő helyzetekben, pl. *p-t, ha p objektum, de p-¿m-et, ha m egy p által mutatott objektum tagváltozója
- Egy pointer különböző időpontokban más objektumokra mutathat
- Pointer értéke bármikor lehet nullptr, amire ügyelni kell (ld. előző pontot)

A referenciák sok szempontból olyanok, mint a pointerek, csak:

- Referencia ugyanolyan szintaxissal hivatkozható, mint maga az objektum
- Referencia mindig arra az objektumra mutat, amelyikkel inicializáltuk
- Nem létezik „null reference", ezért mindig feltételezhetjük, hogy tényleg objektumra mutat

3.2.2. Referenciák típusai

Az egyenlőségek bal oldalán, jobb oldalán használható értékekre való hivatkozások, illetve a konstans értékekre történő hivatkozások megkülönböztetésére 3 referencia-típus létezik:

- **lvalue referencia**: megváltoztatandó objektumokra hivatkozik
- **const referencia**: konstans objektumokra hivatkozik
- **rvalue referencia**: olyan objektumokra hivatkozik, melyek értékét a használat után nem akarjuk felhasználni (pl. temp változók)

Konstans referenciát nagyméretű objektumok esetén szokás használni. Ilyenkor az objektumokat függvényeknek történő átadáskor nem szükséges másolni (ami költséges lenne), viszont garantáltan módosítani sem lehet az értéküket.

3.2.3. Bal oldali (*lvalue*) referenciák

Egy T típusú objektumra mutató *lvalue* referencia: `T&`.

Ha függvénynek referenciát adunk át, a függvény módosíthatja annak a változónak az értékét, amelyre a referencia tulajdonképpen egy alias (tehát úgy is gondolhatunk rá, mint egy const pointerre, amely mindig „dereferenciálódik").

```
int var = 1;
int& r{var}; //lehetne = var is, de a lenyeg: r es var mostmar u.arra
    az intre hivatkoznak
int x = r; //x erteke mostantol 1
r = 2; //var erteke mostantol 2
```

A referenciának muszáj valamire hivatkoznia, tehát nem elég deklarálni, inicializálni is kell:

```
int var = 1;
int& r1{var};
int& r2; //hiba: hianyzik az inicializalas!
extern int& r3; //OK: r3 mashol mar inicializalva volt
```

Mint mondtuk, referencia maga az objektum (nincs rá külön operátor)

```
int var = 0;
int& rr{var};
++rr; //var-t inkrementaljuk 1-re
int* pp = &rr; //pp var-ra mutat
```

`T&` referencia inicializálása csak lvalue-n keresztül tehető meg (olyan objektummal, melynek a címe lekérdezhető):

```
int& r1 = 1; //hiba, mert 1 nem lvalue, ez csak egy ertek
int i = 1;
int& r2 = i; //igy mar mas
```

3.2.4. Konstans referenciák

Konstans T típusú referencia inicializálásakor nem kötelező sem lvalue-t, sem pedig (feltétlenül) pont T típusú értéket megadnunk (jobb oldali érték – vagyis *rvalue* – is használható mint temporary object, és implicit konverzió is létezik pl. `int`-ről `double`-ra):

```
double& dr = 1; //nem OK
const double& cdr {1}; //OK
```

De miért ne lehetne nem const referenciákhoz is temp változót létrehozni? Erre a válasz az, hogy azért, mert ha a referencia értékét módosítanánk, akkor a temp változó értékét is módosítanánk, ami nagyon sok extra könyvelést jelentene a futtatási környezetnek. Ráadásul a C++ szabványban sehol sincs kikötve, hogy hogyan és milyen logika szerint kellene a temp objektumokat tárolni! Összességében tehát a temp objektumok azok azonosító és azonosítható cím nélküli értékek, melyekre nem const referenciával hivatkozni nem lehet.

A const referencia más, mert rajta keresztül az értéket biztosan nem lehet módosítani, csak olvasnunk kell tudni, ugyanúgy mint egy egyszerű jobb oldali érték esetén.

3.2.5. Jobb oldali (*rvalue*) referenciák

Foglaljuk össze az eddig vett referencia-típusokat!

- Non-const *lvalue*: írható referencia
- *const lvalue*: hivatkozható, de nem írható referencia

Az *rvalue* referencia pedig olyan referencia típus, amely *destruktívan olvasott* értékekre vonatkozik. Ez azt jelenti, hogy a referencia létrehozásakor azt állítjuk: az adott objektumot a referencián kívüli más kontextusban már nem fogjuk használni, ezért annak erőforrásai (pl. dinamikus memóriatartalma) másolás nélkül is „ellopható”.

Ez első hallásra bonyolultnak tűnhet, de valójában hatékonyság szempontjából igen hasznos funkció a jobb oldali referencia. Ugyanis ha tudjuk, hogy a hivatkozott adatra más kódrészletben nem lesz már szükség, akkor annak a dinamikus erőforrásait nem szükséges lemásolni, vagyis újra lefoglalni!

Fontos, hogy *rvalue* referencia csak jobb oldali (temp) érték alapján hozható létre, jelölése pedig `T&&`.

```
string var{"Cambridge"};
string f();

string& r1{var}; //lvalue referencia, r1 var-ra hivatkozik
string& r2{f()}; //lvalue referencia, de hiba: f() rvalue-nak szamit!
string& r3{"Princeton"}; //lvalue referencia, de hiba: nem
   hasznalhatunk temp objektumot!

string&& rr1{f()}; //rvalue referencia, es OK: rr1 erteke egy temp
   objektum
//mivel az objektum temp, ezert mashol nem szamithatunk ra es ha
   masolni kell, a futtatasi kornyezet
//donthet ugy, hogy a tartalmat csak "atcimkezi"
string&& rr2{var}; //rvalue referencia, de hiba: lvalue erteket
   adtunk neki
string&& rr3{"Oxford"}; //rendben, rvalue referencia egy temp
   objektumra

const string& cr1{"Harvard"}; //OK, csinalunk egy temp objektumot es
   hozza tartsitjuk cr1-hez
```

Nézzünk még egy példát!

```
std::string f(std::string&& s)
{
  if(s.size()){
    s[0] = toupper(s[0]);
  }
  return s; //amikor itt kimasoljuk, a belso s-t es az eredeti s-t(!)
      mar soha az eletben nem fogjuk hasznalni
}

//igy a fv meghivhato rvalue-val inicializalt stringgel (sot, csak
   azzal):
std::cout << f("muuuukodj!");
```

Előfordulhat néha olyan eset is, hogy a programozó tudja, hogy egy objektumot többet nem fog használni, de a fordító nem. Például:

```
template<class T>
void swap(T& a, T&b) //old-style swap
{
  T tmp{a}; //ezutan ket kopia van a-bol
  a = b; //ezutan ket kopia van b-bol es egy a-bol
  b = tmp; //ezutan ket kopia van tmp-bol (ami valojaban a)
}
```

Ez szuboptimális megoldás, hiszen elegendő lenne a-t és b-t, mint nevet („címkét") felcserélni, igazából teljesen felesleges egy temp változót csak azért létrehozni, hogy át tudjuk mozgatni memória szinten is az a és b-hez tartozó tartalmakat.

Ezért a fentiek helyett megírható a `swap()` függvény az alábbi módon is:

```
template<class T>
void swap(T& a, T&b)
{
  T tmp{static_cast<T&&>(a)}; //az inicializalas irhat a-ba, vagyis a
      -t nem masoljuk, hanem T is a-ra mutat (az a nevet pedig kesobb
      mar nem hasznaljuk ugyanerre)
  a = static_cast<T&&>(b); //az ertekadas irhat b-be, vagyis b-t nem
      masoljuk, csak az a valtozot ugy hozzuk letre hogy a korabbi b-t
       jelentse
  b = static_cast<T&&>(tmp); //az ertekadas irhat tmp-be, ugyanis
      kesobb kulon nem lesz ra szukseg
}
```

Persze könnyű lehet félregépelni a `static_cast<T&&>`-t. Ezért a standard könyvtárnak van egy `std::move()` függvénye, ami ugyanezt csinálja:

```
template<class T>
void swap(T& a, T& b)
{
  T tmp{std::move(a)};
  a = std::move(b);
  b = std::move(tmp);
}
```

A megoldas így majdnem tokeletes, viszont csak lvalue-kra mukodik, pl.

```
void f(vector<int>& v)
{
  swap(v, vector<int>{1,2,3}); //ez nem mukodik mert a 2. argumentum
      nem lvalue
  //ezert vagy tobb move fv-t definialunk, vagy egyeb trukkoket
      alkalmazunk amit most nem reszletezunk
}
```

3.3. Referenciák referenciái

Referencia referenciája is ugyanolyan referencia (tehát nem a referenciára mutató valami, hanem ugyanarra az objektumra hivatkozik). De milyen típusú referencia lesz?

Az ökölszabály az, hogy mindig az *lvalue* referencia „győz”. Magyarán az ún. „reference collapse” szabályok szerint:

- `A& &` eredménye `A&`
- `A& &&` eredménye `A&`
- `A&& &` eredménye `A&`
- `A&& &&` eredménye `A&&`

Ennek gyakorlati jelentősége például az ún. *univerzális referenciálás* esetén van. Ez egy haladó fogalom, de a lényege az, hogy ha egy „külső” függvénynek átadunk egy paramétert, és ezzel a paraméterrel ez a függvény meg szeretne hívni egy másik függvényt, akkor ezt meg tudja tenni függetlenül attól, hogy a második függvény bal vagy jobb oldali referenciát vár. Ez a gyakorlatban úgy néz ki, hogy a „külső” függvényben a paraméter egy generikus típusra hivatkozó jobb oldali referenciaként van definiálva. Amikor egy konkrét típussal átadunk neki egy paramétert, akkor ez vagy `T&`, vagy `T&&` típusú lesz. Mindkét esetben a reference collapse megtörténik, és vagy egy bal oldali, vagy egy jobb oldali referencia marad. Ilyen módon kívülről is vezérelhető, hogy az adott esetben a belsőleg meghívott függvénynek milyen típusú paramétert szeretnénk átadni.

3.4. Gyakorlati tanácsok a referenciák használatát illetően

3.4.1. Mikor érdemes bal oldali referenciát használni?

Ha egy függvénynek bal oldali (és nem konstans) referenciát adunk át, ez azt jelenti, hogy a függvény a saját törzsén belül a külvilág számára is látható módon átírhatja a változó értékét. Ezzel a lehetőséggel csak akkor éljünk, ha a fv. neve kellően beszédes! Például az alábbi kódrészletben elképzelhető, hogy a `next(int p)`, mint szignatúra lehet, hogy beszédesebb, és a hatása jobban érthető / átélhető:

```
void increment(int& aa){
  ++aa;
}

int next(int p){
  return p+1;
}

void g(){
  int x = 1;
  increment(x); //x = 2;
```

```
  x = next(x); //x = 3;
}
```

A bal oldali referenciák visszatérési értékként is hasznosak abból a célból, hogy ez esetben az adott függvény *lvalue* és *rvalue*-ként is használható lesz. Például, egy `Map` típus esetén az indexálás operátort elképzelhető, hogy egy értékadás bal oldalán, illetve olyan is, hogy egy egyenlőségjel jobb oldalán szeretnénk használni. A bal oldali referencia mindkét esetet lefedi, hiszen egy automatikusan dereferenciálódó hivatkozást jelent az adott kulccsal rendelkező értékre:

```
template<class K, class V>
class Map{ //egyszeru Map osztaly
public:
  V& operator[](const K& v); //visszaadja a v kulcshoz tartozo
     erteket
  //ekkor is: mymap[5] = xyz;
  //meg ekkor is: xzy = mymap[5];

  pair<K,V>* begin(){return &elem[0];}
  pair<K,V>* end(){return &elem[0]+elem.size();}

private:
  vector<pair<K,V >> elem; //maguk a kulcs,ertek parok
};
```

Vegyük észre, hogy ha simán `V` típust adnánk vissza, akkor az az adott kulcshoz tartozó érték *másolata* lenne, ami által értékadás bal oldalán a `mymap[5]` kifejezés például nem lenne használható.

3.4.2. *Const lvalue* és *lvalue* referenciák közötti distinkció

Láthattuk, hogy a bal oldali és konstans referencia is inicializálható bal oldali értékkel. Ez értelmezési konfliktusokhoz vezethet: ha egy függvény argumentuma const reference, akkor világos a felhasználás, de mi történik, ha simán bal oldali referenciát vár egy függvény?

Ilyen esetben nem egyértelmű, hogy a függvény deklarációja azért ilyen, mert meg fogja változtatni az argumentum értékét, vagy csak azért, mert nagy objektumokról is szó lehet és nem szerette volna a progrmaozó, ha azt mindig másolni kellene?

Stroustrup tanácsa erre vonatkozóan, hogy mindig legyünk egyértelműek. Ha egy függvény egy értéket nem módosít, az mindig legyen const (ebből nem származik baj, viszont egyértelmű, hogy csak a másolást szeretnénk spórolni). Ha ezt az ökölszabályt betartjuk, akkor az is egyértelmű lesz pusztán a szignatúra alapján, ha egy függvény valóban módosítja egy referencia értékét.

3.4.3. Pointerek és referenciák: Mikor melyiket hasznájuk?

Erre a kérdésre létezik néhány ökölszabály illetve preferencia.

3.4.4. Hivatkozások átirányíthatósága

Ha egy függvényben át szeretnénk irányítani egy hivatkozást másik objektumra, használjunk pointert! Ilyenkor használhatóak a `++`, `--` stb. operátorok is:

```
void fp(char* p)
{
  while(*p)
  {
    std::cout << ++*p;
  }
}

void fr(char& r)
{
  while(r)
  {
    std::cout << ++r; //jaj! a hivatkozott karaktert noveljuk, nem a
       referenciat!
    //majdhogynem vegtelen ciklus
  }
}
```

Ellenben, ha garantáltan mindig ugyanarra az objektumra akarunk hivatkozni, használjunk referenciát! ... Hiszen a referencia inicializálást követően nem változtatható!

```
template<class T>
class Proxy //Proxy az inicializalt objektumra hivatkozik
{
  T& m;

public:
  Proxy(T& mm) : m(mm){}
  //...
};

template<class T>
class Handle //Handle az aktualis objektumra hivatkozik
{
  T* m;

public:
  Proxy(T* mm): m(mm){}
  void rebind(T* mm){m = mm;}
};
```

3.4.5. Operátor-felültöltés referencián

Ha operátor-felültöltéssel szeretnénk műveleteket végezni valamin, ami objektumra hivatkozik, használjunk referenciát (a pointer beépített típus, nem használható op.-felültöltésre)!

```
Matrix operator+(const Matrix&, const Matrix&); //OK
Matrix operator-(const Matrix*, const Matrix*); //hiba: nincs user-
   defined-type argumentum!

Matrix y,z;
//...
Matrix x = y+z; //OK
Matrix x2 = &y - &z; //hiba, es meg ronda is...
```

3.4.6. Kollekciók hivatkozásokkal

Ha olyan kollekciót akarunk, ami objektumokra hivatkozó dolgokból áll, muszáj pointert használni (kivéve, ha az `std::reference_wrapper` osztály template-et használjuk, amely képes referenciákat másolható, értékével átadható objektumba becsomagolni).

```
int x, y;
string& a1[] = {x, y}; //hiba: referenciak tombje nem hozhato letre
string* a2[] = {&x, &y}; //OK
vector<string&> s1 = {x, y}; //hiba: referenciak vektora nem hozhato
   letre
vector<string*> s2 = {&x, &y}; //OK
```

3.4.7. A többi eset

A fentieken túlmenően tényleg ízlés dolga. Ami számíthat, hogy vannak apró különbségek, pl. a pointerek rendelkeznek „nulla” értékkel (`nullptr`), referenciák viszont nem. Például:

```
void fp(X* p)
{
  if(p==nullptr)
  {
    //nincs erteke
  }else
  {
    //hasznalhatjuk *p-t
  }
}

void fr(X& r) //altalanos stilus
{
  //feltetelezzuk, hogy r letezik es hasznalhatjuk
```

```
}
```

Fontos tudnunk továbbá, hogy pointer mutathat pointerre, referencia mutathat referenciára, referencia mutathat pointerre, DE: pointer nem mutathat referenciára – hiszen a referencia nem típus (csak egy név ami valamire utal) és ezért nincs is címe.

Ha pointert adunk át egy függvénynek, a pointert is csak érték szerint adjuk át... ezért ha módosítjuk a pointer címét, annak nem lesz kihatása a hívó környezetre (csak annak lesz, ha a mutatott értéket módosítjuk).

Ezért szoktunk pointerre pointert, vagy pointerre referenciát átadni.

Ebben az esetben például sima pointer nem jó:

```
//globalis valtozo: ezt keruljuk, most csak pelda erdekeben
int g_One = 1;

//fv prototipus
void func(int* pInt);

int main()
{
  int nvar = 2;
  int* pvar = &nvar;
  func(pvar);
  std::cout << *pvar << std::endl; //meg mindig 2

  return 0;
}

void func(int* pInt)
{
  pInt = &g_One;
}
```

Ezért inkább átadhatunk pointerre pointert:

```
//fuggveny prototipus
void func(int** pInt);

int main()
{
  int nvar = 2;
  int* pvar = &nvar;
  func(&pvar);
  //...
  return 0;
}

void func(int** ppInt)
```

```
{
  *ppInt = &g_One; //ppInt mutato erteke most mar g_One-ra mutat

  *ppInt = new int; //igeny szerint meg memoria is allokalhato

  **ppInt = 3; //de mint korabban a mutatott mutato altal mutatott
     ertek is felulirhato
}
```

De ugyanígy használhatunk pointerre hivatkozó referenciát is (ízlés kérdése):

```
//fv prototipus
void func(int* pInt);

int main()
{
  int nvar = 2;
  int* pvar = &nvar;
  func(pvar);
  //...
  return 0;
}

void func(int*& rpInt)
{
  rpInt = &g_One; //rpInt altal hivatkozott mutato erteke most mar
     g_One-ra mutat

  rpInt = new int; //igeny szerint meg memoria is allokalhato

  *rpInt = 3; //de mint korabban a referalt mutato altal mutatott
     ertek is felulirhato
}
```

4. fejezet

Structok és osztályok

> **Forrásmegjelölés**
>
> A fejezetben számos magyarázat és példa Bjarne Stroustrup könyvéből származik: [1].

C++-ban kényelmesen definiálhatunk a beépített típusokhoz hasonlóan funkcionáló *user-defined-type (UDT)* típusokat `struct`-ok és `class`-ok (osztályok) segítségével.

A `struct` és a `class` között árnyalatnyi különbség létezik, ami az alapértelmezett viselkedésükkel kapcsolatos (a tagváltozók láthatósága tekintetében). Ez az árnyalatnyi különbség mégis olyan filozófiai különbséget jelent, ami oda vezetett, hogy a struct-okat inkább összetett (funkcionalitás, metódusok nélküli) adatokra (*Plain Old Data – POD)* szokás használni, míg a class-okat inkább a jól közismert objektum-orientált stílusban szokás használni, ahol az objektumok adatokat és műveleteket is magukban foglalnak.

A kettő közötti (nyelvi) hasonlóság miatt azonban a következőkben összefoglaló néven az *osztály* szót fogjuk használni, akkor is, ha ez struct-ot is jelenthet.

4.1. Osztályok mögötti motiváció, főbb funkciók

Az osztályok használatát az motiválja, hogy számos esetben kell saját típusokat definiálnunk. Tény, hogy a beépített típusok is koncepciók reprezentációi – pl. a `float` típus az operátoraival együtt a matematikai valós számok reprezentációja. Ettől függetlenül a saját alkalmazásainkban általában magasabb szintű fogalmakkal is kell dolgoznunk, pl. egy játék esetén `Explosion` típusokkal, vagy egy szövegszerkesztő esetén `List<Paragraph>` típusokkal.

Az osztályok nemcsak hogy az ilyen típusok létrehozását támogatják, műveleteikkel együtt, hanem a típusok közötti kapcsolatok modellezését is lehetővé teszik (gyűjtő-típus / résztípus, vagy tartalmazó viszony kifejezésére), akár származtatás útján, akár ún. parametrikus polimorfizmus (template változók) útján.

Egy jól megválasztott osztály-hierarchia előnyei, többek között:

- Könnyebben lehet érvelni a program helyességéről
- Általában tömörebb (rövidebb) kód
- Fordító több hibás felhasználást tud detektálni

Egy új típus definiálásakor különválasztjuk egymástól:

- A megvalósítás körülményeihez tartozó részleteket (pl. adatrejtés -– teljesen mindegy, hogy belül mit hogyan reprezentálok!)
- A típus használatához szükséges részleteket (pl. függvények, más néven metódusok listája)

Ebben a C++ nyelv által definiált fájl-típusok is segítenek:

- Header file-ok (.h, .hh, .hpp): típusok deklarálása
- Forrás file-ok (.cpp, .cxx, .cc): típusok megvalósítása

4.2. Miből áll egy osztály

Az osztályok felhaszáló által definiált típusok, melyek:

- Tagváltozókat és tagfüggvényeket tartalmazhatnak
- Tagfüggvényei definiálhatják az inicializálás (létrehozás), másolás, mozgatás és törlés (destrukció) jelentését
- Tagjai objektumok esetén `.` (pont) és mutatók esetén `->` (nyíl) operátorokkal érhetőek el
- Felüldefiniálhatják többek között a +, !, és [] operátorokat
- Saját névteret alkotnak tagváltozóik és tagfüggvényeik tekintetében
- Interfészüket public tagokkal, megvalósítási részleteiket private tagokkal, a származó típusokból látható (de kívülről nem látható) részleteiket pedig protected tagokkal valósítják meg

A `struct` olyan osztály, melynek minden tagja alapértelmezett esetben public. Osztálynak viszont alapesetben minden tagja private.

4.3. Osztály deklarációja és definíciója

Egy osztály deklarációja elegendő ahhoz, hogy a fordító tudja, hogy mi a típus neve és mekkora helyet foglal el a memóriában. Ehhez elegendő megadni az osztály tagváltozóinak típusait, és metódusainak szignatúráit:

```cpp
class X
{
private:
  int m;
public:
  X(int i=0);
  int mf(int i);
}; //pontosvesszo mindig kell, mert nem fv vagy nevter deklaracio

//peldanyositas es felhasznalas
X var(7);

int user(X var, X* ptr)
{
  int x = var.mf(7);
  int y = ptr->mf(9);
  int z = var.m; //hiba: privat adattag nem hozzaferheto
}
```

A példa alapján a fordító tudja, hogy létezik egy `X` nevű típus, amely egyetlen `int` megadásával példányosítható, és a metódusai mellett egyetlen `int` típusú tagváltozót tartalmaz.

A fordító ez alapján ellenőrizni tudja minden fordítási egységben, ahol a deklaráció elérhető, hogy az adott típust megfelelően példányosítjuk-e és megfelelően használjuk-e. Az implementációra majd csak a linkernek lesz szüksége.

Egy osztály definíciója a metódusok definícióját is tartalmazza. Ezek definiálása megtörténhet a `class ... { ... }` deklaráció törzsén belül is – mint lentebb az `mf()` metódus és a fölötte levő konstruktor esetén. Ugyanakkor történhet az osztály deklarációján kívül is – akár külön forrásfájlban is – megfelelő scope-olással, mint az `rf()` függvény esetén.

```cpp
class X
{
private:
  int m;
public:
  X(int i=0) : m(i){}
  int mf(int i)
  {
    int old = m;
    m = i;
    return old;
  }
  int rf();
}; //pontosvesszo mindig kell, mert nem fv vagy nevter definicio

int X::rf() {
    return m;
```

```
}
```

A két megoldás közötti lényeges különbség, hogy a deklaráción belül definiált metódusok esetén a fordító dönthet úgy, hogy azokat *inline*-olja. Ez azt jelenti, hogy a metódus meghívásának összes helyén kicserélheti a függvényhívást a függvény törzsének szó szerinti másolatával, így elkerülhető, hogy futásidőben tényleges függvényhívás történjen, ami általában időben overhead-del jár (új stack frame generálása). Ennek ellenpontja, hogy sok inline-olt hívás esetén a fordító-linker által generált futtatható állomány mérete lesz nagyobb.

4.4. Adatrejtés – struct versus class

Struct esetén alapértelmezettként minden publikus:

```
struct Date
{
  int d,m,y;
  void init(int dd, int mm, int yy); //inicializalashoz
  void add_year(int n);
  void add_month(int n);
  void add_day(int n);
};

void Date::init(int dd, int mm, int yy)
{
  d = dd;
  m = mm;
  y = yy;
}

void Date::add_year(int n)
{
  y += n;
}

// ...
```

Itt hátrány, hogy senki nem mondta meg a fordítónak, hogy csak a tagfüggvények „függhetnek” a dátum reprezentációjától, más ehhez direktben nem férhet hozzá. Így előfordulhat például, hogy később a programozó (vagy egy másik programozó) ilyen kódot ír:

```
void timewarp(Date& d)
{
  d.y -= 200; //hupsz!
}
```

Ezért a structban vagy explicite privátra kell állítanunk a tagváltozókat, illetve azokat a metódusokat, amelyeket kívülről nem szeretnénk elérhetővé tenni, vagy pedig osztályt célszerű használnunk, ahol

alapesetben minden privát (ez esetben viszont a publikus tagváltozókról, illetve metódusokról kell rendelkeznünk):

```
struct Date
{
  void init(int dd, int mm, int yy); //inicializalashoz
  void add_year(int n);
  void add_month(int n);
  void add_day(int n);
private:
  int d,m,y;
};

// vagy:

class Date
{
  int d,m,y;
public:
  void init(int dd, int mm, int yy); //inicializalashoz
  void add_year(int n);
  void add_month(int n);
  void add_day(int n);
};
```

Ezt nevezzük *adatrejtésnek*, és az egyik leginkább evidens előnye, hogy ha egy objektum belső reprezentációja értelmetlen vagy inkonzisztens, akkor arról csakis az osztály megvalósítása tehet. A debuggolás első lépése, a lokalizáció így automatikusan megvalósul.

4.5. Konstruktorok: Objektumok inicializálása

Vegyük észre azonban azt is, hogy az `init()` függvényt nemcsak hogy értelmetlen paraméterekkel is meghívhatjuk akár (15. hó 3000. napja) – amiről az `init()` naív megvalósítása tehet – de el is felejthetjük meghívni!

Ilyenkor nem számíthatunk arra, hogy a memória-címen értelmes érték lesz – így annak olvasása akár a program összeomlását is eredményezi (a fordító sokszor egy értelmes alap-értéket ír a változó helyére, de a lényeg, hogy erre nem számíthatunk).

4.5.1. Alapértelmezett default konstruktor

Megjegyezzük, hogy ez az inicializáló lépés (vagy egy természetes default értékre, vagy csak a szükséges memória-terület lefoglalásával, a memória szemét megtartása mellett) így is, úgy is megtörténik a fenti esetben, mivel a fordító automatikusan generálni fog az osztály számára egy ún. *konstruktor* függvényt, amely még az `init()` meghívása előtt, a változó létrejöttekor a fedél alatt lefut. ***Ezt az alapértelmezett működést úgy tudjuk felülírni, hogy explicite definiálunk legalább***

egy darab saját konstruktort. A fordító ilyenkor kötelez bennünket, hogy valamelyik explicite definiált konstruktort az osztály példányosításakor minden esetben meghívjuk.

4.5.2. Programozó által definiált konstruktorok

Általában véve tehát sokkal jobb az automatikusan generált default konstruktornál, ha saját magunk definiálunk legalább 1 darab értelmes konstruktort! A konstruktor neve ugyanaz kell, hogy legyen, mint az osztályé, és visszatérési típusa nincs:

```
class Date
{
  int d,m,y;
public:
  Date(int dd, int mm, int yy); //konstruktor
};
```

Ezután már kötelező a konstruktort használni!

```
Date today = Date(23, 6, 1983);
Date xmas(25, 12, 1990); //tomor valtozat, ez is OK
Date my_birthday; //hiba: inicializalas hianyzik!
Date masikHiba(10, 12); //hiba: harmadik argumentum hianyzik
Date xmas2{25, 23, 2014}; //ugyanezek mukodnek kapcsos zarojellel is
```

Az inicializálásnak több szintaxisa lehet, de érdemes kapcsos zárójeleket használni (osztályok esetén is, mert a konstruktor inicializálást definiál). Ennek több előnye van:

- Nincs implicit szűkítés („narrowing”). Pl. `char`-bol `int` OK, de forditva nem. `double`-bol sem enged `float`-ot csinálni
- Ritka esetekben nem használhatjuk csak, amikor a kapcsos zarojeleknek más jelentése van. Pl. az `std::vector` osztály esetében alapvetően eltér a zárójel és kapcsos zárójel jelentése (persze a kapcsos zárójel itt is az inicializálásra utal):

```
vector<int> v1(5); //5-hosszu vektort foglalunk le
vector<int> v2{5}; //vektor, melynek elso eleme 5
```

Osztályhoz akárhány konstruktor definiálható, csak az a lényeg hogy különböző számú és/vagy típusú argumentumuk legyen:

```
class Date{
  int d, m, y;
public:
  //...
  Date(int, int, int);
  Date(int, int);
  Date(int);
```

```
  Date(); //default konstruktor: a mai nap
  Date(const char*); //string reprezentacioban megadott datum
};

Date today{4}; //honap es ev a mai nap alapjan
Date july4{"July 4, 1995"};
Date birthday{5,11}; //ev a mai datum alapjan
Date now; //default, mai nap... csak akkor OK, ha van default
   konstruktor
Date start{}; //ugyanugy default
```

Elegánsabb lesz a kód, ha több konstruktor helyett default argumentumokat használunk:

```
class Date
{
  int d,m,y;
public:
  Date(int dd=0, int mm=0, int yy=0); //nagyszeru
  Date(int dd=0, char*); //hiba: ld kesobb
  //...
};

Date::Date(int dd, int mm, int yy)
{
  d == dd ? dd : today.d;
  m == mm ? mm : today.m;
  y == yy ? yy : today.y;
}
```

Default argumentumot sose követhet nem-default (ugyanis hogyan értelmezhetnénk, ha kevesebb paraméterrel hívnánk meg a fv-t?)

A redundancia ellen egy másik lehetőség: tagváltozók inicializálása:

```
class Date
{
  int d {today.d};
  int m {today.m};
  int y {today.y};
public:
  Date(int, int, int);
  Date(int, int);
  Date(int);
  Date();
  Date(const char*);
  //...
};

//eleg csak d-t inicializalni, hiszen minden mast inicializaltunk
```

```
Date::Date(int dd) : d{dd}
{
  //ellenorizzuk, hogy a datum ervenyes-e
}
```

4.5.3. Inicializálás versus értékadás

A C++-ban élesen elkülönül az inicializálás az értékadástól (= operátor), még a konstruktorok esetén is. Ez azt jelenti, hogy van a „0. időpillanatban történő értékadás”, amely konceptuálisan azt jelenti, hogy az objektum már azokkal a tagváltozó-értékekkel jött létre. Minden más értékadás a konstruktor törzsén belül (= operátorral) az már utólagos értékadásnak számít.

Ez a distinkció azért is nagyon lényeges, mert egy adott osztálynak lehetnek olyan tagváltozói, melyeknek értéke konstans, vagy amelyek referenciák, így muszáj nekik kezdeti értéket adni és később nem hivatkozhatnak más objektumra. Az ilyen típusú tagváltozókat (de más tagváltozót is!) az ún. ***inicializáló listában*** inicializálhatunk.

```
class Person {
    std::string name;
    const Date date_of_birth;
    const Person& mother;
    const Person& father;
public:
    Person(std::string nm, Date dob, const Person& m, const Person &f
       ):
        date_of_birth(dob), mother(m), father(f)
    {
        name = nm;
    }
};
```

A fenti példában a név beállítása (`name` tagváltozó) értékadással, vagyis a 0. utáni pillanatban történik. A konstans illetve referencia tagváltozóknak (ez utóbbiak esetében függetlenül attól, hogy konstansok-e) csak a 0. pillanatban adhatunk értékeket, ezt pedig a konstruktor törzse előtt lévő inicializáló listában tehetjük meg.

4.6. Konstans tagfüggvények és `mutable`

A korábbiakban a `const` módosítóról már volt szó. Jelentése: „megígérem, hogy ennek a változónak az értékét nem módosítom”.

Egy `struct` vagy `class`-ban található függvény (metódus) kontextusában a `const` azt jelenti, hogy a metódus garantáltan nem módosítja az osztály tagváltozóinak értékét (és a fordító ezt ki is kényszeríti!)

```
class Date
```

```
{
//...
public:
  int day() const { return d; }
  int month() const {return m; }
  int year() const;
  //...
  void add_year(int n);
};

int Date::year() const
{
  return ++y; //ez hiba, a fordito sem engedi!
}
```

const objektumra / const referenciára nem hívható meg nem konstans metódus (érthető okokból):

```
void f(Date& d, const Date& cd)
{
  int i = d.year(); //OK
  d.add_year(1); //OK

  int j = cd.year(); //OK
  cd.add_year(1); //hiba: konstans valtozo erteke nem modosithato
  //(meg akkor sem, ha a fv amugy nem tenne ilyet!)
  //a fordito nem olyan okos, hogy belenezzen a fuggveny melyebe
  // csak azt nezi hogy const-kent van-e deklaralva!
}
```

Beszélhetünk viszont olyanról is, hogy egy metódus *logikailag* const. Az ilyen metódus felhasználó számára úgy kell, hogy tűnjön, mintha const lenne, de belül mégiscsak muszáj neki (valamilyen okból kifolyólag) az objektum belső állapotát módosítania.

Erre tipikus példa a cache-elt értékek eltárolása. Ha például egy kimenetet pazarló lenne mindig újra és újra „összeállítani”, elképzelhető, hogy az osztály inkább eltárolná az aktuális értékét és csak időnként módosítaná azt.

Tegyük fel, hogy a `Date` osztálynak van egy string reprezentációja, amit költséges lenne mindig újra és újra kiszámítani. Ekkor:

```
class Date
{
public:
  //...
  string string_rep() const; //a reprezentacio lekerdezheto
private:
  bool cache_valid; //ervenyes-e a reprezentacio mostani erteke
  string cache; //maga a reprezentacio
  void compute_cache_value(); //frissitsuk a cache erteket
```

```
  //...
};
```

A felhasználó szempontjából `string_rep()` nem módosítja az objektum állapotát, de a `cache` értékének időnként változnia kell. A probléma, hogy `const` objektum esetén elvileg a `compute_cache_value()` függvényt nem lehetne (még közvetetten, a `string_rep()`-en keresztül sem) meghívni!

Erre egyfajta megoldás lehet, hogy *const_cast* típusú kasztolást használunk. Ez nem szép.

Egy fokkal elegánsabb megoldás a `mutable` módosító használata. Ennek jelentése: ez a tagváltozó `const` objektum esetén is módosítható! Vegyük észre, hogy a lenti kódrészletben így már a technikailag nem konstans `compute_cache_value()` is lehet `const`!

```
class Date
{
public:
  //...
  string string_rep() const; //a reprezentacio lekerdezheto
private:
  mutable bool cache_valid; //ervenyes-e a reprezentacio mostani
     erteke
  mutable string cache; //maga a reprezentacio
  void compute_cache_value() const; //frissitsuk a cache erteket
  //...
};
```

Ezek után:

```
string Date::string_rep() const
{
  if(!cache_valid){
    compute_cache_value();
    cache_valid = true;
  }
  return cache;
}

void f(Date d, const Date cd)
{
  string s1 = d.string_rep();
  string s2 = cd.string_rep(); //OK!!
  //...
}
```

Egy marmadik megoldás a fentebb vázolt problémára, hogy indirekt eléréseket vezetünk be. Ha ugyanis egy objektumnak nemcsak egy kis részét kell módosíthatóvá tennünk, a módosítandó részeket egy külön, pointeren keresztül hivatkozott objektumba tehetjük (a `const` ezekre tranzitívan ugyanis nem vonatkozik):

```
struct cache{
  bool valid;
  string rep;
};

class Date{
public:
  //...
  string string_rep() const;
private:
  cache* c;
  void compute_cache_value() const;
  //...
};

string Date::string_rep() const
{
  if(!c->valid){
    compute_cache_value();
    c->valid = true;
  }
  return c->rep;
}
```

4.7. this kulcsszó, és referencia visszadása metódusból

A C++ nyelven speciális kulcsszó a this, amely minden osztályon vagy struktúrán belül arra az aktuális objektumra mutató pointer, amelyre a metódust meghívtuk (feltéve, hogy nem statikus metódusról van szó!).

Erre a kulcsszóra akkor lehet szükségünk, amikor az adott objektumra hivatkozó pointert vagy referenciát szeretnénk visszaadni. Például:

```
class IntBox {
    int val;
public:
    IntBox(int v) : val(v) {}
    IntBox* increment() {
        val++;
        return this;
    }
    int getVal() {return val;}
};

int main() {
    IntBox ib(5);
    IntBox* pib = ib.increment();
```

```
    if (ib.getVal() == pib->getVal()) {
        std::cout << "Bizony, a ket valtozo erteke ugyanaz!" << std::
           endl;
    }

    if (&ib == pib) {
        std::cout << "Bizony, a ket valtozo cim szerint is ugyanaz!"
           << std::endl;
    }
}
```

A fenti példának jelen esetben túl sok értelme nincs, de működik. Érdekesebb, ha referenciát adunk vissza – ilyenkor ugyanis láncolni is lehet a hívásokat, mivel mindig egy referenciat kapunk vissza!

```
class IntBox {
    int val;
public:
    IntBox(int v) : val(v) {}
    IntBox& increment() {
        val++;
        return *this;
    }
    IntBox& times2() {
        val = val * 2;
        return *this;
    }
    int getVal() {return val;}
};

int main() {
    IntBox ib(5);
    ib.increment().times2().increment();
    std::cout << "ib = " << ib.getVal() << std::endl; // az eredmeny
       13
}
```

Vegyük észre, hogy ez nem a várt eredményt adná, mert minden hívás után az eredeti objektum *másolatát* adja vissza `increment()` és `times2()`. Ezért az első inkrementálás után `ib` másolatát kapjuk vissza, erre hívjuk meg a kétszeres szorzót, majd ez is egy másolatot ad vissza, ahogy az utolsó `increment()` is. A probléma csak az, hogy mi az `ib` értékét írjuk ki, amelyre a legelső hívás még kihatással volt ugyan, de a második, harmadik hívás már egy másolaton, vagy annak a másolatán került meghívásra.

```
class IntBox {
    int val;
public:
    IntBox(int v) : val(v) {}
    IntBox increment() {
```

```
        val++;
        return *this;
    }
    IntBox times2() {
        val = val * 2;
        return *this;
    }
    int getVal() {return val;}
};

int main() {
    IntBox ib(5);
    ib.increment().times2().increment();
    std::cout << "ib = " << ib.getVal() << std::endl; // az eredmeny
       most csak 6...
}
```

4.8. static kulcsszó függvényeken és osztályokon belül

A static kulcsszóval ellátott változó általános esetben előfordulhat osztályban vagy függvényben. E mellett osztály metódusokat is elláthatunk ezzel a kulcsszóval.

A különböző esetek az alábbiakban foglalhatóak össze:

- Ha egy függvényben deklarált és definiált statikus változóról van szó, a változó csak ez első futáskor jön létre és kapja meg a kezdeti értékét. Ezt követően a definíció sorát a program még egyszer már nem hajtja végre. Ezzel a megoldással például megszámolható, hogy egy függvényt hányszor hívunk meg, mint ahogy a lenti példában a `StaticExample` osztály konstruktorában látható. Az ilyen változó tehát a függvény scope-ján belül érhető csak el, viszont mivel az adatszegmensen tárolódik, ezért „túléli” a függvény visszatérését.
- Osztály tagváltozójaként deklarált statikus változót az osztály deklarációján belül a nyelv szabályai szerint nem definiálhatjuk (nem inicializálhatjuk) – ezt minden esetben az osztályon kívül kell megtennünk (mint a lenti példában a `statvar` változó esetében). Az ilyen esetben a változó valójában nem az osztály példányaihoz, hanem az osztályhoz magához tartozik – bizonyos értelemben egy singleton-ként viselkedik.
- Osztály metódusa esetében a static metódus azt jelenti, hogy a metódus az osztályhoz, és nem annak objektumaihoz tartozik (mint a lenti példában a `StaticExample::printMyState()` metódus.

```
class StaticExample {
public:
    static int statvar;
    StaticExample(){
        static int count = 0;
```

```
        count++;
        std::cout << "count is: " << count << " and value of static
           variable statvar is: " << statvar << std::endl;
    };
    static void printMyState() {
        std::cout << "My state: " << statvar << std::endl;
    }
};

int StaticExample::statvar = 1;

int main()
{
    StaticExample obj1;
    std::cout << "\tObj1 created" << std::endl;
    StaticExample obj2;
    std::cout << "\tObj2 created" << std::endl;
    obj1.statvar = 2; // ez es a kovetkezo 2 sor ugyanazt a valtozot
       irja
    obj2.statvar = 3;
    StaticExample::statvar = 4;
    StaticExample obj3;
    std::cout << "\tObj3 created" << std::endl;
    obj3.printMyState();
    StaticExample::printMyState(); // az elozo is mukodik, de
       tulajdonkeppen igy helyes
}
```

5. fejezet

Objektumok másolása, értékadása és mozgatása

5.1. Dinamikus tagváltozók, mint menedzselendő erőforrások

Emlékezzünk vissza, hogy a C++-ban minden olyan adatot, amely nem a *stack*-en, és nem az *adatszegmensen*, hanem a *heap*-en (más szóval: halmon vagy adatszegmensen) tárolódik, azt *dinamikus adatnak* nevezzük.

A stack-re kerülnek a lokális hatáskörű változók, így a stack mérete a függvényhívásokkal nő és visszatérésükkel csökken.

Az adatszegmensre kerülnek a globális illetve a statikus változók.

A heap-et / halmot akkor használjuk, amikor:

- Szeretnénk, ha az adat túlélné' a függvények visszatérését, ugyanakkor mégsem szeretnénk, ha az globális lenne, vagy
- Szeretnénk, ha az adat mérete időben változni tudna (ami a stack-en nem lehetséges)

Például egy láncolt listáról nem tudhatjuk előre, hogy hány eleme lesz. Ezért elemeit a stack-en nem is lenne értelme lefoglalni.

Maga a láncolt lista, mint változó élhet a stack-en (vagyis: megszűnhet ha a stack-ről eltűnik). De az elemeinek számossága nem fix, ezért ha a stack-en levő változó tartalmaz egy pointert a dinamikus memóriára – pl. a lista legelső elemére – és a lista minden eleme magában foglal egy pointert a következő elemre, akkor a lista hossza valójában dinamikusan nőhet / csökkenhet.

Dinamikus tagváltozót a `new` operátor által visszaadott pointerként tárolhatunk el. A láncolt lista példájában tehát az osztály része lenne egy pointer, amelynek értékét (pl. a konstruktorban) ezzel az operátorral állíthatjuk be.

Tudjuk azonban, hogy ha valamikor is `new`-t hívunk meg, `delete`-re is szükségünk lesz! Ezt az osztályban az ún. *destruktor* függvény fogja meghívni – a konstruktor párja, amely az objektum

stack-en való megszűnésekor (vagy ha az objektum maga is a halmon van, akkor az objektumra történő delete meghívásakor) fog lefutni.

5.2. Dinamikus memória lefoglalása konstruktorral és felszabadítása destruktorral

Először vegyük az IntNode osztályt. Egy-egy ilyen objektum fogja a lista elemeit tárolni (az egyszerűség kedvéért a lista elemei legyenek most egész típusú számok):

```
class IntNode {
    int value;
    IntNode* next;
public:
    IntNode(int v) : value(v), next(nullptr) {}
    int getValue() { return value; }
    IntNode* getNext() { return next; }
    void setNext(IntNode* n) { next = n; }
};
```

Az IntLinkedList osztály dinamikusan hoz létre IntNode objektumokat a heap-en:

```
class IntLinkedList {
    IntNode* head;
public:
    IntLinkedList() : head(nullptr) {}
    // ...
    void addNode(int value);
    void print();
};

void IntLinkedList::addNode(int value) {
    if (!head) { head = new IntNode(value); }
    else {
        IntNode* pre = head;
        IntNode* cur = head->getNext();
        while (true) {
            if (!cur) {
                cur = new IntNode(value);
                if (pre) pre->setNext(cur);
                break;
            }
            pre = cur;
            cur = cur->getNext();
        }
    }
}
```

```cpp
void IntLinkedList::print() {
    IntNode* cur = head;
    while (true) {
        if (cur) {
            std::cout << cur->getValue();
        }
        else {
            std::cout << std::endl;
            break;
        }
        cur = cur->getNext();
        if (cur) { std::cout << ", "; }
    }
}
```

Amit dinamikusan létrehoztunk, azt egyszer törölni is kell. Erre való a ***destruktor***, ami ugyanolyan nevű, mint az osztály, csak van előtte egy tilde (~) és nincs visszatérési típusa:

```cpp
class IntLinkedList {
    IntNode* head;
public:
    IntLinkedList() : head(nullptr) {}
    ~IntLinkedList();
    void addNode(int value);
    void print();
};

IntLinkedList::~IntLinkedList() {
    IntNode* cur = head;
    IntNode* next = nullptr;
    while (true) {
        if (cur) {
            next = cur->getNext();
            delete cur;
            cur = next;
        }
        else {
            break;
        }

    }
}
```

Vegyük észre, hogy magát a `delete` operátort csak akkor hívtuk meg, amikor a csomópontot követő következő csomópont címét már eltároltuk a `next` változóba! Ellenkező esetben már azelőtt töröltük volna a változót, hogy tudtuk volna, mi következik utána.

Összefoglalásképpen, tudjuk az alábbiakat:

- Többféle konstruktor létezhet (automatikusan generált default konstruktor, vagy saját magunk által definiált default konstruktor illetve további tetszőleges számú konstruktor, melyek szignatúrája páronként különböző)
- Destruktort muszáj készítenünk, ha az osztálynak van dinamikusan allokált adata. Ha egy osztály neve az, hogy `Something`, akkor:
  - A default konstruktor szignatúrája: `Something()`
  - A destruktor szignatúrája: `~Something()`

5.3. Copy konstruktor

Említettük, hogy ha semmilyen konstruktort nem hozunk létre, a fordító automatikusan generál nekünk egy default konstruktort. Ennek legtöbb gyakorlati esetben nincs sok haszna, ezért érdemes mindig saját konstruktort készíteni.

5.3.1. Automatikusan generált copy constructor

Ugyanígy – alapesetben – a fordító létrehoz nekünk egy ún. ***copy konstruktort*** is! Ez lehetővé teszi, hogy egy adott típusú objektumot egy másik, már létező, ugyanolyan típusú objektum értéke alapján létrehozzunk.

Például:

```
void f() {
    // ll1 a stack-en van, de sok memoriara hivatkozik a heap-en
    IntLinkedList ll1;
    ll1.addNode(5);
    ll1.addNode(6);
    ll1.addNode(7);
    ll1.print();
    IntLinkedList ll2(ll1); // masolat beepitett copy constr-ral
    ll2.addNode(55);
    ll1.print(); // hopp!
}

int main() {
    f(); // kulon f() fuggveny, hogy a destruktort teszteljuk
}
```

Figyeljük meg, hogy egyrészt az utolsó sorban printelt `ll1` is tartalmazni fogja az 55-öt... mivel a másolatban cím szerint ugyanaz a head pointer lesz, és innentől a teljes lista a dinamikus memóriában ugyanarra a memóriaterületre mutat.

Ezután viszont könnyen lehet, hogy a program crash-el (a Visual Studio-ban mindenképpen!). A probléma az, hogy az automatikusan generált copy konstruktor nem olyan „okos”, mint remélnénk!

Ugyanis ez a copy konstruktor a másolatként létrejött új objektumnak bitenként átadja a már létező, lemásolt objektum tartalmát.

Ez esetünkben azt jelenti, hogy a `head` pointer, mint cím értéke ugyanaz lesz a két objektumban, és így a dinamikus memóriában ugyanazt a tartalmat hivatkozzák meg! Ezt ellenőrizhetjük is a pointerek címeinek kiírásával, vagy úgy, hogy az egyik lista értékeit módosítjuk, és megnézzük, mi történt a másikkal (tipp: ugyanaz!).

A default copy constructor működése tehát akkor probléma egyrészt, ha a másolandó objektum dinamikus memóriát (Vagy más erőforrást) használ, és azt szeretnénk, ha a másolat ennek az dupulumát (nem az eredeti hivatkozásokat) kapná meg. Ez önmagában kellemetlen, de a fenti példában a crash akkor történt, amikor egy egyik objektum dinamikus memóriatartalma már felszabadult (a destruktor által), és a futtatási környezet érzékelte, hogy ezáltal a másik – még éppen létező – objektum alól kihúztuk a memóriát.

5.3.2. Saját copy constructor

Ha egy osztály neve az, hogy `Something`, akkor a copy constructor szignatúrája:

`Something(Something&)`

vagy esetleg

`Something(const Something&)`

Az előző példa így már jól működne:

```
class IntLinkedList {
    IntNode* head;
public:
    IntLinkedList() : head(nullptr) {}
    ~IntLinkedList();
    IntLinkedList(const IntLinkedList& other);
    void addNode(int value);
    void print();
};

IntLinkedList::IntLinkedList(const IntLinkedList& other) : head(
   nullptr) {
    IntNode* curInOther = other.head;
    IntNode* preInThis = nullptr;
    while (true) {
        if (curInOther && !head) {
            head = new IntNode(curInOther->getValue());
            preInThis = head;
        }
        else if (curInOther) {
            preInThis->setNext(new IntNode(curInOther->getValue()));
            preInThis = preInThis->getNext();
        }
```

```
        else { break; }
        curInOther = curInOther->getNext();
    }
}
```

5.4. Copy assignment

Az inicializálás (konstrukció) és értékadás (assignment) közötti különbség, hogy értéket adni csak olyan változónak lehet, amely már létezik a memóriában; míg konstrukálni csak olyan objektumot lehet, amely még nem létezett korábban.

Figyelem: itt nem kifejezetten az = operátor használatáról van szó. Objektumot konstrukálni is lehet így:

```
    IntLinkedList ll = IntLinkedList(5);
```

Itt, ha `ll` még nem létezett, akkor ez a hívás a konstruktort fogja meghívni!

Ugyanígy, ha ezt írjuk:

```
    IntLinkedList a(5);
    a.addNode(6);
    IntLinkedList ll = a;
```

Akkor az utolsó sorban hiába használtunk értékadó operátort, valójában a copy konstruktor kerül meghívásra, mivel valamit másolunk egyrészt (a jobb oldalon levő `a` változót), másrészt pedig új változót hozunk létre (`ll`), amely még nem létezett korábban.

A copy assignment operátor akkor hívódik meg, amikor a bal oldalon levő változó sem újkeletű:

```
    IntLinkedList a(5);
    a.addNode(6);
    IntLinkedList ll(10);
    ll = a; // ll erteke most mar ugyanaz, mint a erteke
```

A copy assignment megvalósítása nem-triviális esetben azért bonyolultabb a copy constructorénál, mert nemcsak hogy le kell foglalni a memóriát (erőforrásokat), de a már korábban lefoglalt erőforrásokat fel is kell szabadítani.

A copy assignment működése úgy is felfogható, hogy le kell futtatni a destruktort, és utána a copy constructor-t. De ezt körültekintően kell csinálni, mert a destruktor lehet, hogy olyat is felszabadít, amit nem kéne, vagy nem feltétlenül kéne. Konceptuálisan így érdemes róla gondolkodni.

5.4.1. Automatikusan generált copy assignment operátor

Sajnos (vagy nem sajnos) a fordító copy assignment operátort is generál nekünk automatikusan, ha mi még ezt nem tettük meg! Ez a korábbiakhoz hasonló kellemetlen jelenségekhez vezethet.

5.4.2. Saját copy assignment operátor

Ha egy osztály neve az, hogy `Something`, akkor a copy assignment operátor szignatúrája:

`Something& operator=(Something&)`

vagy esetleg

`Something& operator=(const Something&)`

A copy assignment visszatérési értéke: `*this` – ugyanis a `this` az egy pointer az adott objektumra, és az értékét (referenciáját) szeretnénk visszaadni.

Például:

```
class IntLinkedList {
    IntNode* head;
public:
    IntLinkedList() : head(nullptr) {}
    ~IntLinkedList();
    IntLinkedList(const IntLinkedList& other);
    IntLinkedList& operator=(const IntLinkedList& other)
    void addNode(int value);
    void print();
};

IntLinkedList& IntLinkedList::operator=(const IntLinkedList& other) {
    // azert kell, hogy ne onmagaval tegyuk egyenlove es futtassuk le
        rajta a destruktort
    if (this == &other) return *this;

    // destruktor kodja egy-az-egyben
    IntNode* cur = head;
    IntNode* next = nullptr;
    while (true) {
        if (cur) {
            next = cur->getNext();
            delete cur;
            cur = next;
        }
        else {
            break;
        }
    }

    //DE: hogy ne legyen inkonzisztens allapot, ezt meg kell tennunk:
    head = nullptr;

    IntNode* curInOther = other.head;
    IntNode* preInThis = nullptr;
```

```
    while (true) {
        if (curInOther && !head) {
            head = new IntNode(curInOther->getValue());
            preInThis = head;
        }
        else if (curInOther) {
            preInThis->setNext(new IntNode(curInOther->getValue()));
            preInThis = preInThis->getNext();
        }
        else {
            break;
        }
        curInOther = curInOther->getNext();
    }
    return *this;
}
```

5.4.3. Copy-and-swap idiom

Bár kevésbé hatékony, de sokszor ajánlják az ún. *copy-and-swap* idióma használatát is. Ez azt jelenti, hogy mielőtt a copy assignment operator megvalósításában felszabadítanánk a már létező változó erőforrásait, még azelőtt készítünk egy másolatot belőle. Ezután kicseréljük a másolat belső tartalmát az aktuális objektum belső tartalmával. Az másolt objektum pedig – mivel a stack-en jött létre – automatikusan felszabadul, mihelyt a függvény visszatér (így a destruálással nem is kell foglalkoznunk).

Ennek az az értelme, hogy az erőforrások lefoglalása mindig „nehezebb", mint a felaszabadításuk. Ilyenkor ha netán nem sikerülne az erőforrásokat lefoglalni, ez még azelőtt kiderülne, mielőtt lebontottuk volna a változó korábbi értékét! Tehát a copy-and-swap így nézne ki:

```
IntLinkedList& IntLinkedList::operator=(const IntLinkedList& other) {
    // masolat keszitese:
    IntLinkedList copy_of_other(other);

    // beltartalom kicserelese
    IntNode* tmp_head = copy_of_other.head;
    copy_of_other.head = head;
    head = tmp_head;

    // miutan a fuggveny visszater, copy_of_other destruktora
       automatikusan meghivodik
    // ezaltal a korabbi head (ami most mar copy_of_other.head) es a
       rajta fuggo
    // tovabbi IntNode-ok mind felszabadulnak!
}
```

Ez a megoldás elegáns, konceptuálisan jól érthető. Természetesen nem annyira hatékony, mint a másik megoldás, mivel másolatokat hozunk létre és cserélgetünk ki az aktuális objektummal.

5.5. Rule of 3

A híres „Rule of 3" azt mondja ki, hogy amennyiben egy osztályhoz definiálunk saját destruktort vagy saját copy constructort vagy saját copy assignmentet, akkor a másik kettőt ezek közül szintén célszerű megvalósítanunk.

Az e mögött meghúzódó általános megfigyelés, hogy ezek a metódusok kéz-a-kézben járnak. Nyilvánvalóan ha az egyik esetében nem tudtuk elfogadni az automatikusan generált, triviális megoldást, ennek valamilyen memóriakezeléssel vagy egyéb erőforráskezeléssel (adatbázis handle, szemafor, stb.) kapcsolatos oka lehetett. Ilyenkor pedig nem triviális sem a másolás, sem az értékadás, sem a felszabadítás.

Ha sietünk és nem szeretnénk ezeket megvalósítani, viszont az automatikus legenerálásukat is szeretnénk megtiltani, használjuk a `delete` operátort:

```
    IntLinkedList(const IntLinkedList&) = delete;
    IntLinkedList& operator=(const IntLinkedList&) = delete;
```

Ilyenkor ha csak megpróbálunk másolni vagy értéket adni, a fordító szólni fog, hogy ilyen metódus nem létezik.

5.6. Move constructor és assignmemnt

5.6.1. Mit jelent konceptuálisan a mozgatás?

A C++ működési logikájának alapelve, hogy az adott scope-ban létrehozott (nem dinamikus) változók a stack-re kerülnek, és a scope eltűnésével onnan automatikusan törlődnek (a destruktor meghívásával, ha az adott típusban létezik olyan).

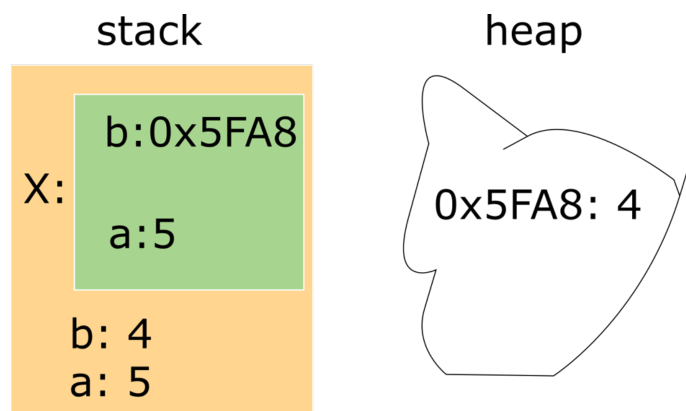
Ezzel együtt a stack-en létrejött változóknak is lehetnek ún. *erőforrásai* – ami takarhat dinamikusan lefoglalt memóriát, adatforráshoz (pl. adatbázishoz vagy file-hoz) tartozó handle-t, esetleg erőforrás exkluzív hozzáféréséhez lefoglalt szemafort.

Az egyszerűbb tárgyalás érdekében maradjunk a dinamikus memóriánál – azonban tudjuk, hogy a következőkben bármilyen erőforrásról lehet szó.

A lenti példában található kód esetében az azt követő ábrán látott helyzet alakul ki:

```
class X {
    int a;
    int* b;
public:
    X(int aval, int bval): a(aval), b(new int(bval)) {}
    ~X() {delete b;}
    void printSum() {std::cout << a + *b << std::endl;}
```

```
};

int main() {
    int a = 5;
    int b = 4;
    X peldany(a, b);
    peldany.printSum();
    // peldany valtozo a stack-en van
    // peldany::a a stack-en van
    // peldany::b a stack-en van
    // *peldany::b a heap-en van!
}
```



Ha a stack-en egy változó megszűnik, de a hozzá tartozó heap memória nincs felszabadítva, ún. *lógó pointer* (dangling pointer) jön létre. Ezért fontos a destructor, amit meg is írtunk: ha egy X típusú változó a stack-en felszabadul, automatikusan meghívódik a destruktora és felszabadul a heap adott memóriaterülete is.

Amikor az objektumot egy másik objektum alapján hozzuk létre (copy constructor), vagy egy másik objektum értékét bele másoljuk (copy assignment), akkor a korábbiakban tárgyaltaknak megfelelően gondoskodnunk kell arról, hogy a dinamikus memória tartalma konzisztens maradjon, ne maradjanak dangling pointerek és minden objektum a saját memóriaterületével rendelkezzen (ha csakugyan ez a cél, hogy ne csak referenciaként funkcionáljanak az adott memóriaterületre!)

Vannak viszont olyan esetek is, ahol tudjuk, hogy az új objektumba (vagy a már létező objektumba) belemásolt tartalom egy olyan objektumból származik, amelyre később már soha nem lesz szükségünk.

Erre tipikus példa, amikor egy függvény / metódus létrehoz egy objektumot, és annak az értékét adja vissza. A C++ ilyenkor ún. Return Value Optimization-t csinál (RVO), ami azt jelenti, hogy az eredményt nem lemásolja a heap vonatkozó tartalmát, hogy a stack-en levő másik változó is tudjon rá hivatkozni, hanem move-olja. RVO viszont csak akkor lehetséges, ha az osztálynak van move constructora / move assignmentje!

Egy másik példa, amikor szimplán egy meglévő változó értéke alapján szeretnénk egy újat létrehozni (vagy egy másik már létező változónak az értéket átadni) úgy, hogy másolás helyett átmozgatjuk a tartalmat.

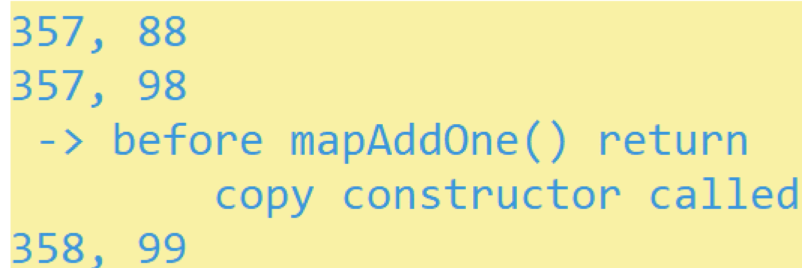
Lényeges, hogy „átmozgatás” alatt itt nem azt értjük, hogy átírjuk egy másik memória rekeszbe, hanem hogy a tartalomra mostantól egy másik változó fog hivatkozni, egy másik változó fogja azt „birtokolni”.

Ezt a célt szolgálja a C++11-es szabvány óta létező *move szemantika*.

5.6.2. Move konstruktor

Tegyük fel, hogy van például egy ilyen metódusunk:

```
IntLinkedList IntLinkedList::mapAddOne() {
    IntLinkedList retval;
    IntNode* cur = head;
    while (true) {
        if (cur) {
            retval.addNode(cur->getValue() + 1);
        }
        else {
            break;
        }
        cur = cur->getNext();
    }
    return retval;
}
```



Ennek használatakor sajnos teljesen feleslegesen a copy konstruktor fog meghívódni, mielőtt a függvény visszatér.

A hatékonyabb működéshez készítsünk move konstruktort! A move konstruktor szignatúrája így néz ki:

`IntLinkedList(IntLinkedList&& other);`

Esetünkben az implementáció viszonylag egyszerű – csak el kell „lopni” a már nem használt láncolt lista címét! Azt viszont ne felejtsük el, hogy a függvény meghívása után other megszűnhet (meghívódhat a destruktora!) Pl. ebben a példában retval meg fog szűnni (a mapAddOne() metódus ha visszatér)! Ezért beállítjuk other.head értékét nullptr-re, így ha delete hívódik rá, nem történik semmi.

```
IntLinkedList::IntLinkedList(IntLinkedList&& other) : head(other.head
   ) {
    std::cout << "\tmove constructor called" << std::endl;
    other.head = nullptr;
}
```

5.6.3. Move assignment

A move assignment szignatúrája így néz ki:

`IntLinkedList& operator=(IntLinkedList&& other);`

Esetünkben a move assignment is nagyon hasonló (hogy melyik hívódik, attól függ, hogy a cél objektum most jön-e létre vagy már létezik). Viszont ilyenkor a már meglévő tartalmat is törölni kell, hogy biztosan ne legyen memóriaszivárgás:

```
IntLinkedList& IntLinkedList::operator=(IntLinkedList&& other) {
    std::cout << "\tmove assignment called" << std::endl;
    // destruktor kodja...
    IntNode* cur = head;
    IntNode* next = nullptr;
    while (true) {
        if (cur) {
            next = cur->getNext();
            delete cur;
            cur = next;
        }
        else {
            break;
        }
    }
    // innentol meg csak atirjuk a pointer cimet:
    head = other.head;
    other.head = nullptr;
    return *this;
}
```

Megjegyezzük, hogy a korszerű fordítók manapság már sokszor move constructor / assignment nélkül is képesek RVO-re... Egyszerűen megoldják, hogy a visszatérő változó értéke már eleve a hívó függvény scope-jában fenntartott változóba íródjon.

Mindez persze nem jelenti azt, hogy haszontalan lenne a move constructor / assignment, hiszen lehetnek olyan esetek, ahol a fordító nem tudja kideríteni, hol jön létre az adott érték, illetve majd látni fogjuk, hogy lesznek esetek, ahol mi magunk szeretnénk majd garantálni, hogy valahol másolás helyett mozgatás történjen.

5.6.4. Move szemantika és kivételek

Végezetül: fontos, hogy a move constructor és move assignment garantáltan ne dobjon kivételt! Gondoljunk bele, mi történne, ha mozgatás közben félúton elhasalna a művelet, és egy catch blokkban kellene folytatnia a programnak a futást.

Ilyenkor már az eredeti objektum se lenne érintetlen állapotban. Ezért, annak érdekében hogy a fordító automatikusan használja a move-ot, sokszor kikötés, hogy a noexcept módosító is szerepeljen a move constructor / move assignment fejlécében:

```
IntLinkedList& IntLinkedList::operator=(IntLinkedList&& other)
   noexcept {
    std::cout << "\tmove assignment called" << std::endl;
    // destruktor kodja...
    IntNode* cur = head;
    IntNode* next = nullptr;
    while (true) {
        if (cur) {
            next = cur->getNext();
            delete cur;
            cur = next;
        }
        // ...
```

5.7. A move szemantika és a referencia típusának kapcsolata

A referenciákban közös, hogy mindegyik egyfajta hivatkozás — a másolást megspóroljuk velük. Említettük, hogy jobb oldali referencia kifejezetten a mozgatást (destruktív olvasást) támogatja, a bal és konstans referencia pedig a költséghatékony átadását értékeknek (úgy hogy írhatjuk, vagy úgy hogy csak olvashatjuk).

De vajon predesztinálva van-e, hogy ha látunk valahol egy jobb oldali referenciát, akkor annak a dinamikus tartalmától búcsút inthetünk? Nem! – ez csak egy lehetőség. Sőt, igazából egy függvénynek átadott bal oldali referenciához tartozó változó beltartalmát is „ellophatja a függvény” – hiszen azt csinál, amit akar, annak adja át, akinek akarja! Magyarán, ha azt a kérdést vizsgáljuk, hogy van-e jelentősége annak, hogy egy függvény bal oldali, jobb oldali vagy konstans referenciát vár, akkor az alábbiakat mondhatjuk:

- Egyrészről (önmagában!) semmi jelentősége nincs... Stílus dolga, illetve annak a kérdése, hogy egy adott függvényt hogyan szeretnénk, hogy meghívjanak: csak bal oldali értékkel, csak jobb oldali értékkel, vagy mindkettővel?
- Persze ha azt mondjuk, hogy az érték ne másolódjon a függvény meghívásakor, akkor csak valamilyen referencia használható (akár jobb, akár bal oldali, akár konstans)
- Fentiektől független kérdés, hogy a függvény saját magán belül mit kezd azzal a változóval. Miután átadtuk, az egy lokális változó (legfeljebb valami olyanra hivatkozik, ami a függvény scope-ján kívül létezik)

- Stilisztikai szempontból lehet értelme, hogy csak akkor használunk egy függvény argumentumban jobb oldali referenciát, ha az átadott hivatkozáson keresztül valóban el is fogja „lopni” majd a változó dinamikus tartalmát
- De ezt semmi nem kényszeríti ki, hogy így legyen! Sőt, akkor is „ellopható” valami, ha bal oldali referenciát adunk rá át, vagy ha előtte lemásoljuk – mivel a C++ eléggé alacsonyszintű nyelv, ez megtehető.

Ezeknek a pontoknak a világosabbá tételéhez vegyünk egy példát!

```
class DynamicInt {
    int* ip;
public:
    DynamicInt(int val) : ip(new int(val)){}

    void print(const std::string& title) {
        if (!ip) {
            std::cout << title << ": nullptr" << std::endl;
        }
        else {
            std::cout << title << ": " << *ip << std::endl;
        }
    }

    DynamicInt(const DynamicInt& other) : ip(new int(*other.ip)) {}

    DynamicInt& operator=(const DynamicInt& other) {
        if (this == &other) { return *this; }
        *ip = *(other.ip);
        return *this;
    }

    DynamicInt(DynamicInt&& other) noexcept : ip(other.ip) {
        other.ip = nullptr;
    }

    DynamicInt& operator=(DynamicInt&& other) noexcept {
        if (this == &other) { return *this; }
        delete ip;
        ip = other.ip;
        other.ip = nullptr;
        return *this;
    }
};
```

Továbbá vegyük az alábbi eseteket!

```
void rvaluePassCopyConstr(DynamicInt&& dint, const std::string& title
   ) {
```

```
    std::cout << "\t" << title << std::endl;
    DynamicInt newdint = dint;
    newdint.print("new int after copy");
    dint.print("old int");
}

void rvaluePassMoveConstr(DynamicInt&& dint, const std::string& title
   ) {
    std::cout << "\t" << title << std::endl;
    DynamicInt newdint = std::move(dint);
    newdint.print("new int after move");
    dint.print("old int");
}

void lvaluePassCopyConstr(DynamicInt& dint, const std::string& title)
    {
    std::cout << "\t" << title << std::endl;
    DynamicInt newdint = dint;
    newdint.print("new int after copy");
    dint.print("old int");
}

void lvaluePassMoveConstr(DynamicInt& dint, const std::string& title)
    {
    std::cout << "\t" << title << std::endl;
    DynamicInt newdint = std::move(dint);
    newdint.print("new int after move");
    dint.print("old int");
}

void copyPassCopyConstr(DynamicInt dint, DynamicInt& dintOutside,
   const std::string& title) {
    std::cout << "\t" << title << std::endl;
    DynamicInt newdint = dint;
    newdint.print("new int after copy");
    dint.print("old int");
    dintOutside.print("original from outside");
}

void copyPassMoveConstr(DynamicInt dint, DynamicInt& dintOutside,
   const std::string& title) {
    std::cout << "\t" << title << std::endl;
    DynamicInt newdint = std::move(dint);
    newdint.print("new int after move");
    dint.print("old int");
    dintOutside.print("original int from outside");
}
```

```
int main() {
    DynamicInt inta(5);
    rvaluePassCopyConstr(std::move(inta), "called
       rvaluePassCopyConstr"); std::cout << std::endl;
    rvaluePassMoveConstr(std::move(inta), "called
       rvaluePassMoveConstr"); std::cout << std::endl;

    std::cout << "------" << std::endl;

    DynamicInt intb(6);
    lvaluePassCopyConstr(intb, "called lvaluePassCopyConstr"); std::
       cout << std::endl;
    lvaluePassMoveConstr(intb, "called lvaluePassMoveConstr"); std::
       cout << std::endl;

    std::cout << "------" << std::endl;

    DynamicInt intc(7);
    copyPassCopyConstr(intc, intc, "called copyPassCopyConstr"); std
       ::cout << std::endl;
    copyPassMoveConstr(intc, intc, "called copyPassMoveConstr"); std
       ::cout << std::endl;
}
```

```
        called rvaluePassCopyConstr
new int after copy: 5
old int: 5

        called rvaluePassMoveConstr
new int after move: 5
old int: nullptr

------
        called lvaluePassCopyConstr
new int after copy: 6
old int: 6
```

```
        called lvaluePassMoveConstr
new int after move: 6
old int: nullptr

------
        called copyPassCopyConstr
new int after copy: 7
old int: 7
original from outside: 7

        called copyPassMoveConstr
new int after move: 7
old int: nullptr
original int from outside: 7
```

Az eredmények között azt láthatjuk, hogy:

- Az eredeti változó dinamikus tartalma túléli azt, ha jobb oldali referenciaként átadjuk egy függvénynek, ami azután copy konstruktorral lemásolja.
- Az eredeti változó dinamikus tartalma természetesen NEM ÉLI TÚL azt, ha jobb oldali referenciaként átadjuk egy függvénynek, ami azután move konstruktorral ellopja.
- Az eredeti változó dinamikus tartalma túléli azt, ha bal oldali referenciaként átadjuk egy függvénynek, ami azután copy konstruktorral lemásolja.

- DE az eredeti változó dinamikus tartalma ugyanúgy NEM ÉLI TÚL azt, ha bal oldali referenciaként átadjuk egy függvénynek, ami azután move konstruktorral ellopja (tehát látható, hogy ez bármilyen referenciánál megtörténhet!).

További érdekesség, hogy ezután a program úgy adja át a változót különböző függvényeknek, hogy átadja értékként is (átadáskor történő másolás), bal oldali referenciaként is, úgy, hogy a függvény aztán a kettő közül az előbbit (a másolatot) belemásolja, vagy (`std::move()` segítségével) átmozgatja egy újonnan létrehozott változóba. Érthető módon ilyenkor az történik, hogy:

- A korábbi esetben a külső változó (amelyre a második argumentum hivatkozik), és az átadott másolat is érintetlen marad, hiszen copy konstruktort használunk
- A második esetben a külső változó szintén érintetlen marad, viszont az átadott másolat már nem, hiszen `std::move()`-ot hívtunk rá még mielőtt az egyenlőség-operátort használtuk volna!

Ez azt is megmutatta, hogy a mozgatást általában az `std::move()` függvénnyel kikényszeríthetjük. Persze végső soron a fordítón múlik, hogy mit fog meghívni. Amennyiben például nincs kikötve, hogy a move konstruktor és move assignment nem dobhat kivételt, dönthet úgy is, hogy nem azokat hívja meg.

6. fejezet

Öröklés

> **Forrásmegjelölés**
>
> A fejezetben számos magyarázat és példa Bjarne Stroustrup könyvéből származik: [1].

Az öröklés (származtatás) célja, hogy a felhasználó által definiált típusok között hierarchikus viszonyokat hozzunk létre:

- Ha egy osztályt már megírtunk, felmerülhet az igény, hogy annak egy specializáltabb típusát is létrehozzuk
- Ha egy osztályt már megírtunk, felmerülhet az igény, hogy annak implementációját egy másik osztályban újra-hasznosítsuk, de egyedi interfészt hozzunk neki létre (más interfészbe csomagoljuk)

Mindkét eset egyfajta kód-újrafelhasználást jelent, és mindkét esetre megoldást jelent a származtatás.

6.1. Specializált típusok

Ha azt mondjuk, hogy „öröklés” vagy „származtatás”, legtöbb embernek a specializált típusok jutnak eszébe.

Az OOP-ben a világban előforduló dolgokat (objects) és azok képességeit modellezzük. De minden dolognak lehetnek altípusai és általánosabb megvalósulásai is.

Például minden autó egyben jármű is. Viszont a Ferrari egyfajta autó.

Ugyanígy, minden alkalmazott egyben ember is (ma még). De egy alkalmazott lehet menedzser, gyakornok vagy éppen adminisztrátor is.

6.1.1. A specializált típusok mögötti motiváció

Mi az alábbi kóddal a probléma?

```
class Employee {
    std::string name;
    int id;
public:
    Employee(std::string s, int i): name(s), id(i) {}
};

class Manager {
    Employee* emp;
    std::string department;
public:
    Manager(Employee* e, std::string d) : emp(e), department(d) {}

};
```

A probléma kettős:

1. Mi értjük, hogy `Manager` is egyfajta `Employee`, de a fordító ezt nem tudja, csak azt látja, hogy van rá egy hivatkozás
2. Ezért nem fog szólni nekünk, ha valahol (bárhol) úgy használunk egy menedzser objektumot ahogy egy alkalmazottat nem kéne, és nem is engedélyezi, hogy bármely kódrészletnek, amely `Employee` objektumot vár, `Manager` objektumot adjunk át (noha az is egyfajta alkalmazott)

Például jó lenne, ha ezt meg lehetne tenni:

```
void print_employee_name(Employee* e) {
    std::cout << e->name << std::endl;
}

Employee x("Szabo Istvan", 25);
Manager xm(&x, "HR");

print_employee_name(&xm); // hiba!xm is egy Employee, csak
   tortenetesen Manager...
// de ezt a fordito nem tudja
```

De itt az is látszódik, hogy teljesen feleselegesen létre kellett hoznunk egy külön `Employee` típusú objektumot, hogy a címét átadhatssuk a `Manager` konstruktorának.

Ha ezt a `Manager` konstruktorában foglalnánk le dinamikusan, akkor már a destruktora se lenne triviális…

6.1.2. Megoldás: publikus származtatás

Inkább származtassuk a `Manager` osztályt az `Employee` osztályból formálisan is:

```
class Employee {
    std::string name;
    int id;
public:
    Employee(std::string s, int i): name(s), id(i) {}
};

class Manager : public Employee {
    std::string department;
public:
    Manager(std::string s, int i, std::string d) : Employee(s, i),
       department(d) {}

};
```

Vegyük észre az alábbiakat!

- A származtatás úgy történik, hogy az osztály neve után kettősponttal elválasztva leírjuk, hogy `public Employee` (ahol `Employee` helyére mindig az ősosztályt írjuk. Hogy ez miért `public`, arról később lesz bővebb szó;
- `Manager` osztály most már látszólag nem tartalmaz hivatkozást `Employee` osztályra, viszont származik belőle;
- Az ősosztályt (`Employee`) a gyermek osztály konstruktorának inicializáló listájában inicializálhatjuk valamely konstruktorának meghívásával. Fontos, hogy ha az ősosztálynak nincsen automatikusan generált default konstruktora (mert létezik legalább 1 programozó által definiált konstruktora), akkor abban az esetben, ha a programozó nem definiált default (üres) konstruktort, **kötelező** a gyermek osztály inicializáló listájában valamelyik nemtriviális konstruktort meghívni!

6.1.3. Származtatott típusok memória-szerkezete

Fontos tudni, hogy a motorháztető alatt a származtatott osztály összetevődik:

- Egy a szülő méretének megfelelő összefüggő memóriaterületből
- És a fentihez hozzávett további, csak a származtatott osztályra jellemző adatokból

Mivel a szülő „rész” összefüggő, így a specializáltabb „részeket” a futtatási környezet mindig a kezdőcímhez képesti offsettel találja meg.

Fentiekből következik egyrészt az is, hogy a származtatott objektum mérete sohasem lehet kisebb, mint a szülőé.

De ami még fontosabb, hogy ha egy interfészen egy a szülőnek megfelelő típusú objektum (címét, vagy referenciáját) várjuk, mégis egy származtatott típusú objektum címét (vagy referenciáját) adjuk meg neki, nincsen probléma! Mivel a kettő címe egy és ugyanaz! (csak az objektum végét kéne levágni, hogy a szülő típusú objektumot kapjuk, de transzparens módon kezelhetjük a címet úgy is, mintha csak a szülő objektumnak megfelelő adatok lennének ott).

Ezért például most már gond nélkül megtehetjük, hogy:

```
void print_employee_name(Employee* e) {
    std::cout << e->name << std::endl;
}

Manager x("Szabo Istvan", 25, "HR");

print_employee_name(&x); // nem ihba, mert az &x cimen levo adatok
   tortenesen egy Employee adataival kezdodnek
// Persze, print_employee_name nem tamaszkodhat olyanra, amely csak a
    Manager tipusban talalhato meg
```

Ugyanez működik referenciákkal is:

```
void print_employee_name(Employee& e) {
    std::cout << e.name << std::endl;
}

Manager x("Szabo Istvan", 25, "HR");

print_employee_name(x); // no problemo
```

Ami viszont nem működik, az a sima objektumok átadása:

```
void print_employee_name(Employee e) {
    std::cout << e.name << std::endl;
}

Manager x("Szabo Istvan", 25, "HR");

print_employee_name(x); // hiba! x nem egy Employee tipus
```

Ennek oka, hogy az `Employee` és `Manager` típusoknak a memóriában más a méretük! A függvény pedig azzal számol, hogy a stack-en egy `Employee`-nak megfelelő memóriaterületet foglal le – amihez képest `x` nagyobb!

A korábban ismertetett felcserélhetőség tehát csak pointerekre és csak referenciákra működik, mert ezek mérete a memóriában fix (a pointer egy cím, ami a gép memóriaméretétől függ, hogy mekkora helyet foglal el, de típustól független).

6.2. Láthatóság változásai örökléskor

A származtatott osztály ugyanúgy használhatja a szülő *public* és *protected* adattagjait és metódusait, mintha a sajátjai lennének.

Ami viszont a szülőben *private*, azt nem érheti el a gyermek osztály. Ez a fajta változó is ugyanúgy megtalálható a gyermek osztály memóriaterületén, de a fordító nem engedélyezi, hogy írjuk vagy olvassuk.

Tehát az eddigiek alapján:

- Osztály privát tagja nem érhető el sem a származtatott osztályból, sem kívülről
- Osztály publikus tagja elérhető kívülről és származtatott osztályból is

A kettő elegye a protected, ami kívülről nem érhető el, de származtatott osztályból igen.

Ez azonban még csak a kezdet, mert a C++ magához a származtatáshoz is láthatóságot társít!

6.2.1. Publikus öröklés

Ahogy említettük, *is-a* jellegű specializiált típust *public* származtatással kell létrehozni (amikor azt mondjuk, hogy a származtatott típus a szülő típus **egy fajtája**):

```
class Employee {
    std::string name;
    int id;
public:
    Employee(std::string s, int i): name(s), id(i) {}
};

// public jelentese: Manager EGYFAJTA Employee
class Manager : public Employee {
    std::string department;
public:
    Manager(std::string s, int i, std::string d) : Employee(s, i),
       department(d) {}

};
```

Ezt úgy kell olvasni, hogy „***Manager egyfajta Employee***."

Publikus örökléskor minden, ami a szülőben public, illetve protected, az a gyermekben is public, illetve protected lesz. Ami a szülőben private, az a gyermekben nem látszódik.

6.2.2. Protected öröklés

Protected örökléskor minden, ami a szülőben public, illetve protected, az a gyermekben is protected lesz. Ami a szülőben private, az a gyermekben nem látszódik.

Ennek a fajta öröklésnek akkor van értelme, ha egy már létező megvalósítást „be szeretnénk csomagolni", és egy új interfészt szeretnénk neki adni.

Ami az ősosztályban akár publikus is volt, az a gyermek osztályban már protected, tehát kívülről nem elérhető! Viszont a gyermekosztály készíthet újabb publikus interfészt, amit kiajánl a programozók számára.

6.2.3. Privát öröklés

Privát örökléskor minden, ami a szülőben public, illetve protected, az a gyermekben privát lesz. Ami a szülőben private, az a gyermekben nem látszódik.

Ennek a fajta öröklésnek egyrészt szintén akkor van értelme, ha egy már létező megvalósítást „be szeretnénk csomagolni", és egy új interfészt szeretnénk neki adni – ugyanúgy, mint a protected öröklés esetén – csak vegyük észre, hogy private öröklés esetében már nincs lehetőség a további származtatásra, hiszen a gyermekosztályban (legalábbis az eredeti, „nagyszülő" osztályból) már semmi sem látszódna.

Egy másik alkalmazási területe a privát öröklésnek a birtoklási viszonyok (*has-a* típusú kapcsolatok) reprezentálása. Ha egy `Car` osztály privát módon örököl a `Battery` típusból, és egy `VendingMachine` osztály is örököl a `Battery` típusból, ezzel nem azt fejezzük ki, hogy mindkettő egyfajta akkumulátor, hanem azt, hogy mindegyiknek van egy akkumulátora.

Ilyenkor azonban van más lehetőségünk is: egyszerűen tartalmazhat mindkét osztály egy `Battery` típusú tagváltozót – így a privát öröklés felhasználási köre valójában elég limitált, és OOP szempontból nem is túlságosan érdekes.

6.3. Összefoglaló példa

Az alábbi példa minden örökléssel / láthatósággal kapcsolatos esetre mutat példát:

```
class A {
public:
    int x;
protected:
    int y;
private:
    int z;
};

class EgyfajtaA : public A {
    // x public
    // y protected
```

```
    // z nem elerheto innen
};

class WrapperA : protected A {
    // x protected
    // y protected
    // z nem elerheto innen
public:
    std::vector getXY() {
        // uj interfesz!
    }
};

class HasAnA : private A {
    // x private
    // y private
    // z nem elerheto innen
public:
    print() {
        std::cout << "Yuhuu, " << x << "es" << y << "a
           rendelkezesemre all";
    }
}
```

7. fejezet

Dinamikus polimorfizmus

Forrásmegjelölés

A fejezetben számos magyarázat és példa Bjarne Stroustrup könyvéből származik: [1].

A származtatott osztály maga is lehet további osztályok szülője.

Az így létrejött osztályhierarchia általában fa struktúrájú, de előfordulhatnak más gráf-típusok is. Az ösöklések láncolata felírható irányított, aciklikus gráfként.

Például:

```
class Employee {};
class Manager: public Employee{};
class Director: public Manager{};

class Temporary {};

class Assistant: public Employee {};
class Intern : public Temporary, public Assistant {};
class Consultant: public Temporary, public Manager {};
```

Látható, hogy adott osztály akár több szülő tulajdonságait is örökölheti (a C++ nem olyan, mint a Java vagy C#, ahol csak egy osztályból – és akárhány interfészből – lehet örökölni).

7.1. Dinamikus polimorfizmus problémafelvetése

Ebben a fejezetben két fő kérdést tekintünk át:

1. Ha egy függvény / metódus kap egy `Base*` típusú változót, hogyan tudja eldönteni, hogy azon a címen valójában milyen konkrét osztály példánya található?

2. Ha ebben az esetben meghívja a pointer által hivatkozott objektum egy metódusát, hogyan lehet elérni, hogy a megfelelő konkrét osztály metódusa hívódjon meg?

Erre a kérdésre 4-féle választ adhatunk:

1. Elkerüljük ennek a lehetőségét. Minden függvény argumentuma csak egyféle típusra mutató pointer lehet (ez OOP szempontból nem praktikus)
2. Hozzunk létre egy típusmezőt a szülőosztályban, amit minden gyermek írhat / és megvizsgálhatunk (ronda és könnyen hibához vezet)
3. Használjunk virtuális függvényeket
4. Használjunk dynamic castot

Ezek közül a második, harmadik és negyedik opciót *dinamikus polimorfizmusnak* nevezzük, mert futásidőben dől el, hogy adott memóriacímet vagy adott függvényt milyen típus szerint használunk (melyik típus azonos nevű metódusát hívjuk meg).

7.2. Egy álmegoldás: strucc-politika!

Egy lehetséges megoldásként említettük: „Minden függvény argumentuma csak egyféle típusra mutató pointer lehet”

Az objektum-orientált programozás pont arról szól, hogy a világban levő dolgokat és a köztük levő hierarchikus viszonyokat hatékonyan ábrázoljuk / kihasználjuk. Ha van egy hangfájl és egy képfájl, mindkettő egy fájl, ezért az alapvető fájlműveletek mindegyikre alkalmazhatóak kell, hogy legyenek.

Ha van egy `open(File*)` függvény, annak el kell tudnia döntenie, hogy milyen fájl-típusról van szó (vagy a megfelelő osztály `open()` metódusát meg kell tudni hívnia).

Az nem hatékony megoldás, hogy 5-féle `open()` függvényt valósítunk meg – mindegyiket más fajta fájlra specializálva.

7.3. Egy másik álmegoldás: típusmezők használata

Kerüljük a típusmezőket! Ugyanis mi a probléma ezzel?

```
struct Employee {
    EmplType type;
    string name;
public:
    enum EmplType {man, empl};
    Employee() : type{empl} {}
    string getName() {return name;}
};
```

```
struct Manager : public Employee {
    vector<Employee*> group;
public:
    Manager() {
        type = man;
    }
};

// mostantol a fordito ellenorzese nelkul garazdalkodhatunk
void print_employee(const Employee* e) {
    switch(e->type) {
        case Employee::EmplType::empl:
            std::cout << name << std::endl;
        case Employee::EmplyType::man:
            std::cout << name << std::endl;
            const Manager* mp = static_cast<const Manager*>(e);
            std::cout << "Manages:" << std::endl;
            for (Employee* ep : mp->group) {
                std::cout << ep->getName() << std::endl;
            }
    }

}
```

Az ordító probléma ezzel, hogy mihelyt új típusú alkalmazottat hozunk létre, módosítani kell az `Employee` osztályt (enum bővítése), és módosítani kell ezt a függvényt is. Ez ellentmond a híres SOLID elvek közül a másodiknak (***open-closed principle***) – miszerint az osztályok zártak a módosításra, és nyitottak a kiterjesztésre.

A szép megoldás az lenne, ha mindegyik osztály maga rendelkezne arról, hogy hogyan kell kiiratni a részleteit, és ha a külső függvényben transzparens módon meg lehetne hívni a megfelelő változatot attól függően, hogy pontosan milyen objektumról is van szó!

7.4. Jó megoldás: virtuális függvények, absztrakt osztályok

Az OOP-ben alapvető technika, hogy a szülőosztály metódusait a gyermekosztály felülírhatja, átértelmezheti.

7.4.1. Virtuális függvények

Ha egy metódust a szülőosztályban virtuálisként jelölünk meg (`virtual`), pontosan azt mondjuk a fordítónak, hogy „***Ez a metódust a származó osztályokban felüldefiniálható***”.

Vegyük észre, hogy ez egy lehetőség (felüldefiniálHATó), nem előírás. A származó osztály meg is tarthatja a szülő osztálybeli implementációt.

Tekintsük meg az alábbi példát!

```
struct Employee {
protected:
    string name;
public:
    Employee(string n) : name(n) {}
    virtual void print() {
        std::cout << name << std::endl;
    }
};

struct Manager : public Employee {
    vector<Employee*> group;
public:
    Manager(string n) : Employee(n) {}
    void addManagee(Employee* ep) { group.push_back(ep); }
    // override nem kotelezo de hasznos lehet:
    void print() override {
        std::cout << name << std::endl; // lehetne Employee::print()
            is!
        std::cout << "Manages:" << std::endl;
        for (Employee* ep : group) {
            ep->print();
        }
    }
};

// mostantol a fordito tudja, hogy milyen tipusu a pointer valojaban!
void print_employee(const Employee* e) {
    e->print();
}
```

Vegyük észre, hogy sok tekintetben tisztább kódot kaptunk:

1. `print_employee()` meghívásakor a futtatási környezet automatikusan tudja, hogy az argumentum milyen konkrét típus, és az annak megfelelő `print()` metódust fogja meghívni
2. Ennélfogva nincs szükség statikus kasztolásra sem (nem mi vagyunk okosak, hanem a futtatási környezet)
3. Nincs szükség típusmező menedzselésére, ami kézileg történne és el lehetne írni / félre lehetne gépelni

Lényeges, hogy a `virtual` kulcsszó egy lehetőséget ad, nem kötelez semmire. Ha a `Manager` osztály nem akarná, nem lenne muszáj felüldefiniálni a `print()` metódust!

Az `override` kulcsszó pont emiatt hasznos: ha nem írnánk oda a felüldefiniáló metódus szignatúrája után, és véletlenül félregépelnénk a metódus nevét, vagy argumentum listáját, a fordító nem szólna, hogy nem override-olunk semmit!

```
struct Employee {
protected:
    string name;
public:
    Employee(string n) : name(n) {}
    virtual void print() {
        std::cout << name << std::endl;
    }
};

struct Manager : public Employee {
    vector<Employee*> group;
public:
    Manager(string n) : Employee(n) {}
    void addManagee(Employee* ep) { group.push_back(ep); }
    // override-dal a fordito szol, hogy elgepeltunk valamit, ez a fv
        semmit sem override-ol!
    //void prit() override {

    // hupsz: azt hittem, override-olom
    void prit() {
        std::cout << name << std::endl; // lehetne Employee::print()
           is!
        std::cout << "Manages:" << std::endl;
        for (Employee* ep : group) {
            ep->print();
        }
    }
};

// mostantol a fordito tudja, hogy milyen tipusu a pointer valojaban!
void print_employee(const Employee* e) {
    e->print(); // meglepetes!
}
```

7.4.2. Virtuális függvények működési mechanizmusa

Érdemes érteni, hogy hogyan működik a `virtual` kulcsszó a fedél alatt.

Ha a fordító látja ezt a kulcsszót, akkor a helyére egy pointer-nyi memóriaterületet foglal le, amelyik az objektumnak megfelelő implementált függvényre fog mutatni. Az implementált függvény egy ún. virtuális táblában (*vtable*) kap helyet.

Kapóra jön, hogy – ahogy említettük – származtatott objektum példányosításakor mindig előbb `A` legősibb felmenő-osztályának megfelelő memóriaterület kerül lefoglalásra. Például, ha van egy `C` osztályunk, amely származik `A` `B` osztályból, amely származik az `A` osztályból, akkor egy `C` típusú objektum memóriaterülete felfogható úgy, mint egy nagy doboz, melynek `A` tetején ott van egy A-méretű rekesz; az alatt egy további rekesz amely az A-hoz képest csak `A` B-ben levő adatokat tartalmazza, míg legalul csak `A` kifejezetten C-ben speciálisan létező adatokat tartalmazó rekesz következik.

Ha most az `A` osztályban van egy virtuális függvény, akkor a legelső rekeszben helyet kap egy pointer (akár `A`-t, akár `B`-t, akár `C`-t példányosítjuk), amely a megfelelő implementációra fog mutatni. A fordító pedig a példányosítás pillanatban pontosan tudja, hogy melyik konkrét típust példányosítottuk!

Innentől kezdve a meghívás transzparens módon történhet, csak végig kell követni, hogy az objektum első rekeszében fix helyen levő pointer milyen függvényre mutat. Ezért jó, hogy ez a pointer a legelső rekeszben kap helyet, mert ez a pointer konrkét típustól függetlenül az objektum kezdőcíméhez viszonyítva ugyanott fog elhelyezkedni.

7.4.3. Absztrakt osztályok

Végezetül fontos tudni, hogy magát a virtuális függvényt sem kell mindig az ősosztályban definiálni. Amennyiben a függvényt *pure virtual*-ként deklaráljuk – ami úgy történik a gyakorlatban, hogy 0-ával tesszük egyenlővé – akkor nem kell definiálni:

```
struct Employee {
protected:
    string name;
public:
    Employee(string n) : name(n) {}
    virtual void print() = 0;
};
```

Azt az osztályt, amelynek létezik legalább 1 *pure virtual* metódusa, **absztrakt osztálynak** nevezzük.

Az ilyen osztály nem példányosítható, hiszen legalább az egyik metódusához nem tartozik implementáció!

Fontos: ez nem jelenti azt, hogy egyébként ne lehetnének az osztálynak adatai, megvalósított metódusai. Ezeket felhasználhatjuk, ha származtatunk belőle és a leszármazó osztály minden pure virtual metódust megvalósít.

Összességében: csak az az osztály példányosítható, amelyben minden függvény implementálva van, és amely az összes, közvetlen ősosztályában (ősosztályaiban) levő pure virtual függvényt implementál.

Ritkán fordul elő, de hasznos tudni azt is, hogy az osztályhierarchiában bármely osztály dönthet úgy is, hogy egy korábban implementált metódust mégis pure virtuallá tesz. Felüldefiniálja úgy, hogy nincs implementációja. Ezért fontos az, hogy a hierarchia egyel korábbi szintjén levő osztályokban levő összes pure virtual metódust definiálnia kell a gyermek osztálynak ahhoz, hogy ne legyen absztrakt, és következésképpen példányosítható legyen.

7.5. Dinamikus polimorfizmus dynamic_cast útján

Sokan nem szeretik, nem tartják elegáns funkciónak a dynamic_cast-ot. De a nyelv lehetővé teszi, hogy dinamikusan kasztoljunk, és vannak esetek, amikor véleményem szerint mégis hasznos dolog, hogy létezik.

7.5.1. A dynamic_cast mögötti motiváció

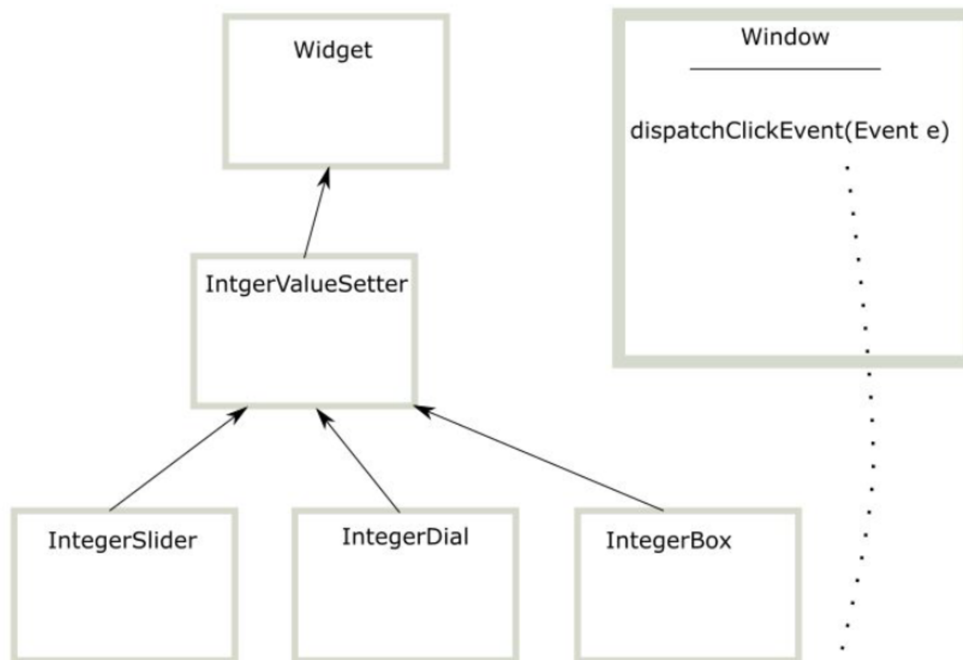
Ha egy osztályhierarchiában több szinten ugyanazok a nevű és paraméter-listájú metódusok fordulnak elő, akkor a virtuális metódusok OOP-szempontból remek megoldást jelentenek.

De mi történik, ha egy függvénynek attól függően más és más további függvényeket / metódusokat kell meghívnia, hogy éppen milyen konkrét típussal dolgozik?

Mint mondtuk, ilyen esetekben a típusmező nem ideális megoldás. Részben megoldást jelenthetne egy `typeof()` virtuális metódus, de csak látszólag: ügyelni kéne arra, hogy az idők végezetéig minden származtatott osztály egyedi azonosítót adjon vissza, és tesztelni kéne a kimenetre, ami sok hiba forrása lehetne.

Amikor meg kell tudnunk, hogy milyen konkrét típussal van dolgunk, használjunk inkább dinamikus kasztolást!

Példa: Tegyük fel, hogy egy GUI-t fejlesztünk. Egész számok megadására a GUI sokféle interfészt kínál: van benne csúszka, tekerentyű, pöcök, stb. Ezen interfészek ősosztálya az `IntegerValueSetter` . Ez az ősosztály pedig a `Widget` ős-ősosztálytól örököl, mivel ő egyébként egy widget is. Van egy ablakozó osztályunk is - `Window` - ami a widgeteket kirajzolja. Ha egy csúszkát (`IntegerSlider`) állítunk, a `Window` osztály kódja meghívja a `myEventHandler(Widget* pw)` függvényt. A fő kérdés: hogyan döntse el a `myEventHandler` függvény, hogy amit kapott, az milyen értéket állít?



Válasz: A C++-ben a `dynamic_cast` segítségével „kibarkochbázhatjuk”, hogy adott változó milyen típusú.

```
void myEventHandler(Widget* pw) {
    IntegerValueSetter* pivs = dynamic_cast<IntegerValueSetter*>(pw);
    if (pivs != nullptr) {
        int x = pivs->getValue();
    }
    else {
        // hupsz! varatlan esemeny
    }
}
```

Itt megjegyezzük, hogy a példában teljesen mindegy, hogy konkrétan milyen módon állítottuk az `int` típust (csúszkával? pöcökkel? mindegy...). Az `IntegerValueSetter` és a `Widget` is két interfész, és közöttük tudunk konvertálni.

A dynamic_cast hívás kacsacsőrök között (template paraméterként) vár egy pointer vagy referencia típust, argumentumként pedig egy konkrét pointert vagy referenciát.

A template típus azt mondja meg a fordítónak, hogy milyen típusra szeretnénk konvertálni az adott pointert vagy referenciát.

Ha a konverzió sikertelen, a dynamic_cast:

- pointer-argumentum esetén nullptr-t ad vissza
- referencia argumentum esetén bad_cast kivételt dob (olyan ugyanis van, hogy pointer nem mutat semmire, de egy referenciának mindig hivatkoznia kell valamire... mivel null referencia azonban nincs, ezért jobb híján kivétel keletkezik)

Összességében, mivel a nullptr hamis érték, a fenti példa tömörebben is kifejezhető:

```
void myEventHandler(Widget* pw) {
    if (IntegerValueSetter* pivs = dynamic_cast<IntegerValueSetter*>(
       pw)) {
        int x = pivs->getValue();
    }
    else {
        // hupsz! varatlan esemeny
    }
}
```

Referencia esetén pedig nincs mást mit tennünk, mint hogy try ... catch blokkba tesszük az utasításokat:

```
void myEventHandler(Widget& pw) {
    try {
        IntegerValueSetter& pivs = dynamic_cast<IntegerValueSetter&>(
           pw);
        int x = pivs->getValue();
    }
    catch (std::bad_cast& ex) {
        // hupsz! varatlan esemeny
    }
}
```

8. fejezet

Standard Template Library alapok

https://en.wikibooks.org/wiki/C

8.1. A Standard Template Library története

1993-ban Alexander Stepanov bemutatott egy új könyvtárat az ANSI/ISO C++ szabványtestületének - amit Standard Template Libary (STL)-nek neveztek. A könyvtár fogadtatása kitörően pozitív volt.

1994 nyarára Stepanov és Meng Lee által összeállított javaslatok belekerültek a C++ szabvány vázlatába (draft version). Augusztusra a Hewlett Packard is felkarolta a projektet, és az ő megvalósításuk (amiben Stepanov is részt vett) széleskörben elterjedt.

A végleges szabványba soha nem került bele, de a C++ Standard Library igen, aminek erős alapot ad az STL. A C++ SL sem nem rész-, sem pedig nem bővített halmaza az STL-nek, de rengeteg a közös pont bennük. A köznyelvben ma is sokan mondanak STL-t C++ Standard Library helyett.

Érdemes nekünk is használnunk a Standard Template Library nevet - bár tudnunk kell, hogy itt az eredetinek arra a részére gondolunk, ami be is került a C++ Standard Library-be.

Azért is érdemes az STL-hez mint névhez ragaszkodunk, mert a C++ Standard Library-ben más célú osztályok / függvények is helyet kapnak, de a mai órán inkább az STL-lel közös részről lesz szó.

8.2. Mi az a template?

Mire jók a template-ek, milyen célt szolgálnak?

A template-ek osztályok, függvények és egyéb nevek (aliasok) típussal történő paraméterézést teszik lehetővé, hogy ne kelljen többször ugyanazt lekódolni.

Például egy vektor tárolhat inteket meg stringeket is, de egyébként belül a két esetben ugyanúgy működik – felesleges lenne tehát kétféle vektor osztályt készíteni.

Általános koncepciók tömör reprezentálására is jók a template-ek (például: lehet egy template osztályt készíteni ami bármilyen típussal működik, ami támogatja a szorzat operátort, vagy valami más operátort.

Sok esetben gondolhatunk a template-ekre úgy, mintha típus-absztrakciók lennének. Az adott template működése csak azoktól a közös dolgoktól függ, amiket bizonyos típusokból meg akarunk ragadni.

Másszóval: a közös dolgokon túl semmi egyéb kapcsolatot nem feltételezünk közöttük. E tekintetben a template-ek merőben eltűnnek a származtatás mechanizmusától, és használatukat ezért hívjuk *dinamikus polimorfizmus* helyett ***parametrikus polimorfizmusnak***.

Pédául: Azon kívül, hogy a mátrixok és vektorok is skalár-szorozhatóak, semmilyen kapcsolat nem kell hogy közöttük legyen ahhoz, hogy általános skalárszorzat-függvényt írjunk (nem kell egymásból sem származniuk).

A template-ek egyik fő előnye, hogy a tervezőnek nem kell lekötnie, hogy csak adott típus használható, más nem.

A kód így rugalmasabb lesz, hiszen nem kell n különböző típusra n-szer (majdnem) ugyanazt lekódolni.

8.2.1. Mire ügyeljünk, amikor template-et használunk?

Ugyanakkor template-ek készítésekor pár dologra oda kell figyelnünk:

- Ha olyan paraméterrel példányosítunk egy template-et, ami nem teljesít minden követelményt (pl. a template osztály egyik metódusa meghív egy olyan másik függvényt, ami nem értelmezett a típusra), akkor a hiba nem a példányosításkor, hanem felhasználáskor fog keletkezni. Ez megnehezíti a programozó dolgát, amikor olyan hibaüzeneteket lát, amelyhez látszólag nincsen köze és nem magyarázzák el jól, hogy melyik template-példányosításnál nem megfelelő a használat!
- A másik, hogy template osztály (vagy függvény) önmagában nem egy teljes osztály (vagy függvény), hanem inkább egy mintázat arra, hogyan kell egy konkrét osztályt (vagy függvényt) legenerálni (amikor már tudja a fordító, hogy milyen típussal paraméterezzük).

Ez a második pont azt is jelenti, hogy template osztály (vagy függvény) megvalósítása sosem lehet külön fordítási egységben (külön cpp fájlban) – csak include-olt header file-ban. Ugyanis ha a main.cpp-ben készítek egy ilyet, hogy `std::vector<int> x;` – akkor a fordítónak hozzá kell férnie a teljes implementációhoz (nemcsak a deklarációhoz), hogy az adott osztályt legenerálja és tudja, hogy milyen típusok, metódusok vannak benne!

8.2.2. Hogyan készíthetünk template típust?

Template változó deklarálható `template<typename C>` formátumban, vagy a `template<class C>` formátum is elfogadott – a kettő ugyanaz.

Álljon itt néhány példa:

```
template <class C>
```

```
int f(C a) {
    return a + 1;
}

template <> // mivel itt C tipusa std::string, uresen hagyhatjuk a
   template argot
int f(std::string s) {
    return 0;
}

int main() {
    int* k = new int{5};
    std::cout << f(5) << std::endl; // 6
    std::cout << f<std::string>("hah") << std::endl; // 0
    delete k;
}
```

Vagy egy template osztály:

```
template <typename T>
class ValuePrinter {
    T val;
public:
    // minden metodust a .h fajlban implementalni is kell!!
    ValuePrinter(T v) {
        // azert, mert amikor peldanyositjuk, be kell helyettesiteni
           a tipust mindenhova
        // de a fordito egyszerre csak egy cpp fajlt lat!
        val = v;
    }
    void print();
};

template<typename T>
void ValuePrinter<T>::print() {
    // minden T tipusra jo, amely ostream operatorral hasznalhato
    std::cout << "value is: " << val << std::endl;
}

// Az osztaly adott tipusra specializalhato is!
template<>
class ValuePrinter<char> {
    char val;
public:
    ValuePrinter(char v) : val(v) {}
    void print() {
        std::cout << "Character value: " << val << std::endl;
    }
```

```
};
```

8.3. Az STL szerkezete

Térjünk vissza a Standard Template Library-re! 4 fő komponenst érdemes megkülönböztetni:

1. Containers (kontérek) - értékek általános tárolására
2. Iterators (iterátorok) - konténerekben levő értékek általános iterálására
3. Functions (függvények) - gyakran újra és újra megvalósított függvények
4. Algorithms (algoritmusok) - felhasználási mintázatok, mint rendezés, keresés, stb.

8.4. Containerek

Attól függően, hogy milyen struktúrában tárolunk adatokat, más és más típusú konténert érdemes használni.

A szekvencia-konténerek valamilyen sorrendbe rendezett adatok tárolására jók.

Ilyen container a `vector`, `list`, `deque` (ejtsd: dekk - jelentése: double-ended queue), `array`, `forward_list` .

Ezeknek az osztályoknak mind nagyon hasonló az interfészük, de vannak finomságbeli különbségek. Például a `list` osztály egy duplán láncolt lista (mindegyik elem tartalmaz hivatkozást a következő és előző elemre), ezért a beszúrás nagyon gyors, de az a random access lassú. `vector` esetén pont a beszúrás lassabb lehet, ha éppen át kell méretezni a vektort mert betelt a lefoglalt memória! (vagy azért, mert át kell helyezni minden elemét, amik elé be szeretnénk szúrni).

8.4.1. Példa: `std::list`

Magyarázó példa a lista típushoz:

```
std::list<int> mylist = {1,2,3,4,5};
// nincs indexalas, csak vegig menni lehet rajta:
// std::cout << "Element 3 of list is " << ????
// mylist[3] nem mukodik!

// viszont: konnyu beszurni
mylist.insert(
    std::next(mylist.cbegin(), 3),
    10 // 4. elem ele beszurjuk hogy 10
);
```

```
std::cout << "Elements of list are: ";
for (int i: mylist) {
    std::cout << i << ", ";
]
std::cout << std::endl;
```

8.4.2. Példa: Fák bejárása `std::deque`-kel

Csak, hogy ezzel az osztállyal is megismerkedjünk: a `deque` sok szempontból hasonlít a vektorhoz, de nem folytonos mem.területen tárolja az elemeit (folytonos részletekben tárolja).

A vektor végére általában konstans a beszúrás (kivéve, ha újra kell méretezni a vektort, és akkor ráadásul minden már meglevő elemét új helyre fogja másolni a futtatási környezet) – ezért azt mondják, hogy amortized constant a költsége.

Egy `deque` elejére és végre is nagyon gyorsan be lehet szúrni. Átméretezéskor nem másol semmit, csak újabb területeket foglal le. Viszont az indexálás kicsit lassabb, mivel nem teljesen folytonos.

A `deque` kiváló eszköz a fabejáráshoz. Fa bejárásánál a látogatott listának mindig egyik végéről szedjük ki a következő elemet, és annak gyermekeit egyik vagy másik végére tesszük.

Ha a gyermekeket ugyanarra a végére szúrjuk be, ahonnan kiolvasunk, akkor mélységi bejárást kapunk. Ha az ellenkező végére, akkor szélességi bejárást.

```
std::deque<Node*> visitedNodes = {root};
while (visitedNodes.size() > 0) {
    Node* node_to_process = visitedNodes.back();
    node_to_process.print();
    visitedNodes.pop_back();

    for (Node* ch : node_to_process->getChildren()) {
        visitedNodes.push_front(ch); // szelessegi bejaras
        // vagy: push_back? - akkor m lys gi bejaras
    }
}
```

8.4.3. Asszociatív konténerek

A szekvencia-konténerek mellett érdemes ismerni az asszociatív konténereket is. Ezekre példa a `set` és a `map`. Az ilyen konténerek értékeket illetve érték-párokat tárolnak rendezett módon. A rendezettség megkönnyíti az érték szerinti keresést.

```
typedef std::pair<int, std::string> jatekos;

std::list<jatekos> psg = { jatekos(7, "Mbappe"), jatekos(9, "Cavani")
   };
psg.push_front(jatekos(1, "Buffon"));
```

```
psg.push_back(jatekos(11, "Di Maria"));

// Set: rendezett elemek, egy elem csak 1x szerepelhet benne
std::set<jatekos> psgJatekosok;
psgJatekosok.empty() ? std::cout << "Meg nincs jatekosunk" : std::
   cout << "Mar vannak jatekosok";
std::cout << std::endl;

for (jatekos j : psg) {
    psgJatekosok.insert(j);
}

psgJatekosok.empty() ? std::cout << "Meg nincs jatekosunk" : std::
   cout << "Mar vannak jatekosok";
std::cout << std::endl;

for (jatekos j: psgJatekosok) {
    std::cout << j.first << ": " << j.second << std::endl;
}
std::cout << std::endl;

// map: kulcs-ertek parok
// a kulcsok rendezettek igy logaritmikus eleresu a kereseskor
// egy kulcshoz csak 1 ertek tartozhat
std::map<jatekos, std::string> jatekosToCsapat;
jatekosToCsapat.insert(std::make_pair(jatekos(1, "Buffon"), "PSG"));
jatekosToCsapat.insert(std::make_pair(jatekos(6, "Pogba"), "
   Manchester United"));
jatekosToCsapat[jatekos(7, "Mbappe")] = "PSG";

if (jatekosToCsapat.find(jatekos(6, "Pogba")) != jatekosToCsapat.end
   ()) {
    std::cout << "Pogba csapata: " << jatekosToCsapat[jatekos(6, "
       Pogba")] << std::endl;
}

for (std::pair<jatekos, std::string> pair : jatekosToCsapat) {
    std::cout << pair.first.second << " csapata: " << pair.second <<
       std::endl;
}
```

8.5. Iterátorok

Az STL általános interfészt ad konténerek elemeinek iterálására / címzésére. Ezekre már láttunk több példát.

Az alapelv: tejesen mindegy, hogy egy konténer belül hogyan tárol elemeket, végig kell tudni rajtuk iterálni.

Minden konténer osztály névterében van egy iterator nevű belső osztály. Ilyen iterátort általában le lehet kérni a begin(), end() vagy cbegin(), cend() metódusokkal (utóbbi esetekben konstans iterátort kapunk, amin keresztül értéket nem lehet módosítani).

Az iterator nemcsak iterálásra, hanem pozíció megjelölésre is jó. Pl. insert() esetében megmondja, hova szúrjunk be. Vagy find() esetében visszaadja a keresett elem pozícióját (amennyiben nem találjuk, az end()-del lesz azonos, így tesztelhető a címek azonossága).

Az end() egyébként az utolsó utáni elemre mutató cím, így nem keverhetjük össze a két esetet, amikor az utolsó elemet kerestük, vagy a keresett elem nincs benne a struktúrában.

8.6. Függvények és algoritmusok (<functional> és <algorithm> header)

A függvény objektumok olyan objektumok, amelyek megvalósítáják az operator() metódust, ezért meghívhatóak ugyanúgy, mintha függvények lennének. Az ilyen objektumokat functoroknak is szokták nevezni.

Példa kézileg létrehozott functorra:

```
struct mystruct {
    int operator()(int a) {return a;}
} myobject;

int x = myobject(28); // x erteke 28
```

A <functional> headerben sok hasznos függvény elérhető functor formában. Ennek az a fő előnye hogy ezeket a függvényeket átadhatjuk más függvényeknek argumentumként anélkül, hogy meg kéne őket írni.

Az <algorithm> pedig olyan gyakran használt eljárások vannak, mint páronkénti művelet-végrehajtás, feltételes összeszámolás, szűrés, és még rengetegféle.

Például:

```
#include <iostream>
#include <functional> // std::plus miatt
#include <algorithm> // std::transform miatt

int main() {
    int first[] = {1,2,3,4,5};
    int second[] = {10,20,30,40,50};
    int results[5];
    std::transform(first, first+5, second, results, std::plus<int>())
        ;
}
```

Fenti példában azt kértük a számítógéptől, hogy vegye a `first` első 5 elemét, és ezeket adja össze a second azonos számú elemével páronként, az eredményeket pedig írja a `results` tömbbe.

Az összeadás maga egy reifikált (tárgyasult) függvény, melyet az `std::plus<int>` típus példányosításával adunk meg.

Egy másik példa:

```
#include <iostream>
#include <functional>
#include <algorithm>

int main() {
    int numbers[] = {20, -30, 10, 40, 0};
    int cx = std::count_if(
        numbers,
        numbers + 5,
        std::bind2nd(std::greater_equal<int>(), 0)
    );
    std::cout << "There are " << cx << " non-negative elements" <<
       std::endl; // 3-at ir ki
}
```

Itt összeszámoltuk a feltételnek megfelelő elemeket.

9. fejezet

Design patternek a C++ nyelvre adaptálva

Forrásmegjelölés

A fejezetben számos magyarázat és példa az alábbi weboldalról származik:

`https://en.wikibooks.org/wiki/C%2B%2B_Programming/Code/Design_Patterns`

A ***design pattern*** olyan absztrakció, amely mintázatot ad egy rendszer struktúrájának felépítéséhez

A híres "Gang of Four" nyomán alakult ki ez a kifejezés (Gamma, Helm, Johnson, Vlissides) – akik 1994-ben publikálták a „*Design Patterns: Elements of Reusable Object-Oriented Software*" című könyvüket.

Nagy hatású könyv volt, sokan ma is esküsznek a design patternekre. Természetesen kritikusai is vannak ennek a megközelítésnek, ahogy a fejezet végén látni fogjuk. Ettől függetlenül azért érdemes őket ismerni és ahol hasznos, felhasználni őket.

9.1. A design patternek fő típusai

Háromfajta design patternt különböztetünk meg:

- Creational patterns (kreációs minták) – arra adnak választ, hogy hogyan hozzunk létre objektumokat (akkor érdekes, ha pl. futásidőben dől el, hogy pontosan milyen típusú objektumot szeretnénk; vagy ha több lépcsőben kell szabályozni egy példány inicializációját és ezt káoszos lenne 1 konstruktorral megtenni)
- Structural patterns (strukturális minták) – entitások funkcióinak bővítése, hogy más entitásokkal kompatibilisek legyenek, vagy azokban foglalt funkciókat (is) megvalósítsák

- Behavioral patterns (viselkedési minták) – objektumok közötti kommunikáció hatékony megvalósításai, akár indirekt úton is!

9.2. Kreációs minták

9.2.1. Builder minta

Probléma: összetett objektumot szeretnénk létrehozni, de nem akarjuk, hogy ehhez összetett (sokargumentumos) konstruktort, vagy több inicializáló függvényt kelljen meghívni

A megoldás: köztes (Builder) objektum létrehozása, mely kiterjeszthető interfészt ad a létrehozandó objektum különböző részeinek inicializálásához.

Példa: egy pizzának többféle tésztája, feltéte lehet. Ahelyett, hogy mindezen információkat a `Pizza` osztály konstruktorának kellene betáplálni, hozzunk létre:

- A `Pizza` osztályon belül mindenféle settert (`setDough()`, `setTopping()`, ...)
- Egy `PizzaBuilder` absztrakt osztályt, mely kiajánl egy megfelelő interfészt (`getPizza()`, `buildDough()`, `buildSauce()`, `buildTopping()`)!
- Ebből származtatva létre lehet hozni a konkrét pizza builder, azaz a `HawaiianPizzaBuilder`, `SpicyPizzaBuilder`, ...stb. osztályokat, melyek mind egy speciális `Pizza` objektumot hoznak létre és setterekkel beállítják a megfelelő értékeket.
- A `PizzaMaker::makePizza(PizzaBuilder*)` metódusa pedig végig hívogatja a megfelelő `build ...` metódusokat.

9.2.2. Factory minta

Probléma: Futásidőben szeretnénk eldönteni, hogy pontosan milyen objektumot hozunk létre. A fordítás idejében ezt még nem tudjuk.

Megoldás: Készítsünk egy Factory osztályt, melynek `New(const std::string&)` statikus metódusa létrehoz egy konkrét példányt. Mivel a `New` metódus csak egy típusra hivatkozó pointert tud visszaadni, ez a típus legyen egy absztrakt ősosztálya a lehetséges típusoknak.

Példa: Legyen `Computer` absztrakt ősosztály. Ebből származnak a `Laptop`, `Desktop` osztályok. A `ComputerFactory` osztály statikus `NewComputer(const std::string&)` metódusa pedig egy pointert, `Computer*`-t ad vissza, mely cím egy `new`-val létrehozott `Laptop` vagy `Desktop` típusú változó.

Fontos, hogy mivel ezeket a pointereket `new`-val hozzuk létre, egyszer majd még `delete`-et is kell hívni rájuk. Ez a hívó fél felelőssége!

9.2.3. A Builder és Factory minták összehasonlítása

A Builder minta esetében a Builder osztályból volt többféle változat (pl. a `HawaiianPizzaBuilder`, `SpicyPizzaBuilder`), melyek ugyanazon `Pizza` osztály settereit hívogatták.

A Factory minta esetében viszont egyetlen Factory osztály volt, viszont annak egy (static) metódusa az argumentuma függvényében különböző típusú objektumokat hozott létre. Annak érdekében, hogy ezek címét visszaadhassa, a típusoknak közös (absztrakt) őse kellett, hogy legyen.

A két módszer persze részben átalakítható egymásba, pl. lehetett volna úgy is csinálni, hogy a `Pizza` osztályból örököl a `HawaiianPizza` osztály meg a `SpicyPizza` osztály, majd a `PizzaFactory` osztály `NewPizza()` metódusa egy `Pizza*` címet ad vissza. Persze ezt a címet a hívó félnek majd egyszer fel is kell szabadítania, ha már nem használja.

9.2.4. Singleton minta

Probléma: Garantálni szeretnénk, hogy adott osztálynak csak egyetlen példánya létezzen az egész alkalmazásban. Tipikusan a menedzser osztályok ilyenek (adatbázis menedzser, hozzáférés-menedzser, ...).

Megoldás:

- A `SingletonX` osztályban a konstruktor privát, a copy constructor és copy assignment pedig le vannak tiltva – ezzel garantáljuk, hogy az osztály csak belülről példányosítható
- A `SingletonX` osztálynak van egy statikus `SingletonX` típusú, `instance` nevű változója
- A szintén statikus, de publikus `SingletonX::GetInstance()` metódus egy `SingletonX&` referenciát ad vissza, mely erre a statikus változóra hivatkozik.
- Vegyük észre, hogy kívülről csak a `GetInstance()` lesz elérhető. Ez azért statikus, hogy úgy is meghívható legyen, hogy az osztályt nem példányosítjuk (az osztályhoz tartozik, nem a példányhoz) – hiszen amúgy példányosítani nem is tudnánk!
- Ugyanígy az instance változó is pont azért statikus, hogy az osztályhoz tartozzon, ne annak egy példányához.

A megvalósítás tehát:

```
class IntegerSingleton {
    int value;
    IntegerSingleton() : value() {}
    IntegerSingleton(const IntegerSingleton& other) = delete;
    IntegerSingleton& operator=(const IntegerSingleton& other) =
       delete;
    static IntegerSingleton* instance;
public:
    void setValue(int v) { value = v; }
        static IntegerSingleton& GetInstance() {
```

```
        // static IntegerSingleton* instance = new IntegerSingleton;
           // ezzel 1 sor megsporolhato
        return *instance;
    }
};

IntegerSingleton* IntegerSingleton::instance = new IntegerSingleton
   (0);

int main() {
    //IntegerSingleton is1 = IntegerSingleton::GetInstance(); //
       nincs copy constructor
    IntegerSingleton::GetInstance().setValue(5);
    IntegerSingleton::GetInstance().print();
    IntegerSingleton::GetInstance().print();
    IntegerSingleton::GetInstance().setValue(6);
    IntegerSingleton::GetInstance().print();
    std::cout << "address of singleton: " << &IntegerSingleton::
       GetInstance() << std::endl;
    std::cout << "address of singleton: " << &IntegerSingleton::
       GetInstance() << std::endl;
}
```

9.3. Strukturális minták

9.3.1. Adapter minta

Probléma: A szoftver egyik komponense egy adott interfészt biztosító objektumokkal működik, de mi egy másmilyen intefészt biztosító objektummal is használni szeretnénk.

Megoldás:

- Ha egy `function(InterfaceTypeA*)` függvényt szeretnénk használni egy `InterfaceTypeB` interfészt megvalósító osztállyal, akkor annyi a teendőnk, hogy egy `Adapter` osztályt készítünk az `InterfaceTypeB`-t megvalósító típushoz.
- Ennek az `Adapter` osztálynak a konstruktora vár egy `InterfaceTypeB*`-t, amit az osztály eltárol.
- E mellett az `Adapter` osztály megvalósítja az `InterfaceTypeA`-t is (pl. származik belőle, vagy csak ugyanolyan szignatúrájú metódusokat tartalmaz). Ezen metódusok megvalósításai az eltárolt `InterfaceTypeB*` címen levő objektum dolgait hívogathatják. . .

A megvalósítás tehát:

```
class Printable {
```

```
public:
    virtual void print() = 0;
};

class Person : public Printable {
public:
    void print() { std::cout << "I am a person" << std::endl; }
};

class Animal {
public:
    void saySomething() { std::cout << "Hi I am an animal!" << std::
        endl; }
};

void printSomething(Printable* prp) {
    prp->print();
}

class AnimalPrintableAdapter : public Printable {
    Animal* animal;
public:
    AnimalPrintableAdapter(Animal* ap) : animal(ap) {}
    void print() { animal->saySomething(); }
};

int main()
{
    Person p;
    printSomething(&p);
    Animal a;
    // printSomething(&a); // animal nem egyfajta Printable
    AnimalPrintableAdapter apa(&a);
    printSomething(&apa);
}
```

9.3.2. Decorator minta

Probléma: Dinamikusan szeretnénk több osztályhoz további funkciót / viselkedést hozzácsatolni, ahelyett, hogy egyenként kiterjesztjük azokat. Ha például egyazon funkciót 25 osztályhoz hozzá akarnánk csatolni, nyögvenyelős lenne 25 különböző osztályból származtatni csak az adott funkcióért!

Megoldás:

- Az egész működés előfeltétele, hogy a „dekorálandó” osztályoknak legyen egy közös public őse

- Ebből a közös ősből származik szintén egy új `Decorator` osztály, amely az ős bizonyos virtuális metódusait pure virtuálissá alakítja át (igen, ilyet lehet csinálni, ahogy korábban is említettük!)
- A `Decorator`-ból származó konkrét `DecoratorX`, `DecoratorY` stb. osztályok pedig ezeket a metódusokat újradefiniálhatják. A decorator osztályok emellett tárolnak egy hivatkozást egy dekorálandó példányra is.
- Mivel a dekorálandó és decorator osztályoknak is van közös őse, gyakorlatilag felhasználhatóak ugyanazokon a helyeken. Egy `DecoratorX` típusú objektum pl. használható egy dekorált típus helyett, miközben maga is tartalmaz önmagában egy ilyen típusú objektumra hivatkozást.

Tehát például:

```
class Car {
    std::string description;
public:
    Car(const std::string& desc) : description(desc) {}
    virtual void getDescription() { std::cout << description << std::
       endl; }
};

class AudiA6 : public Car {
public:
    AudiA6() : Car("Audi A6") {}
};

class RenaultClio : public Car {
public:
    RenaultClio() : Car("RenaultClio") {}
};

class AbstractCarDecorator : public Car {
protected:
    Car* mycar;
public:
    AbstractCarDecorator(Car* cp) : Car(""), mycar(cp) {}
    void getDescription() = 0;
};

class CarWithSoundSystem : public AbstractCarDecorator {
public:
    CarWithSoundSystem(Car* cp) : AbstractCarDecorator(cp) {}
    void getDescription() {
        mycar->getDescription();
        std::cout << "\tThis car also has a sound system!" << std::
           endl;
    }
};
```

```
class CarWithAC : public AbstractCarDecorator {
public:
    CarWithAC(Car* cp) : AbstractCarDecorator(cp) {}
    void getDescription() {
        mycar->getDescription();
        std::cout << "\tThis car also has an air conditioner!" << std
           ::endl;
    }
};

int main()
{
    AudiA6 audi1;
    AudiA6 audi2proto0;
    CarWithAC audi2(&audi2proto0);

    RenaultClio rc1, rc2proto0;
    CarWithAC rc2proto1(&rc2proto0);
    CarWithSoundSystem rc2(&rc2proto1);

    audi1.getDescription();
    audi2.getDescription();
    rc1.getDescription();
    rc2.getDescription();

    // itt mindossze arra kell figyelni, hogy audi2proto0 meg
       rcproto0 es rcproto1
    // ne tunjenek el memoriabol meg azelott, hogy audi2 es rc2
       eltunnenek!

    // Pl.:
    std::cout << "Problem:" << std::endl;
    AudiA6* audi3proto0 = new AudiA6;
    CarWithAC audi3(audi3proto0);
    audi3.getDescription();
    delete audi3proto0;
    audi3.getDescription(); // jo esetben crash!

    // Erre nyujthat megoldast a smart pointerek hasznalata, mint std
       ::unique_ptr
    // Ez tulmutat a targy keretein, de az a lenyege, hogy ha
       audi3proto0 egy
    // std::unique_ptr lenne, akkor az audi3 letrehozasakor "
       atruhazhato" lenne
    // ennek a pointernek a birtoklasa az audi3 reszere ->
    // innentol audi3 erteke nullptr es csak CarWithAC destruktora
       tudna felszabaditani.
```

```
    // https://stackoverflow.com/questions/26318506/transferring-the-
       ownership-of-object-fromone-unique-ptr-to-another-unique-ptr-i
}
```

9.4. Viselkedési minták

9.4.1. Command minta

Az OOP-ben (és a programozásban általában) célszerű nem előre behuzalozni, hogy ki kommunikálhat kivel.

Különösen az OOP-ben előfordulhat, hogy változik, mennyire specializált osztály szeretne mennyire általános interfészhez hozzáférni.

Funkcionális programozásban az ún. callback mechanizmus támogatja, hogy egy függvény lefutását követően paraméterezhető legyen, hogy mi történjen

Ezt C++-ban is meg lehet csinálni. A command pattern osztályok szintjén tesz lehetővé valami hasonlót.

Probléma: bizonyos funkcionalitások lefutását követően további műveleteket szeretnénk dinamikusan végrehajtani

Megoldás: reprezentáljuk magát a kérést egy objektummal, amelyet később átadhatunk a különböző függvényeknek, és amely alapján el tudják dönteni, mi legyen a következő művelet.

Konkrétabban:

- A `Command` osztályunk egy absztrakt osztály, melyből további, specializáltabb parancs-típusok származhatnak. Legegyszerűbb esetben a `Command` osztálynak van egy `execute()` pure virtual metódusa.
- Ezen parancs-típusok példányaira a hivatkozásokat (pointereket, referenciákat) ezután szabadon lehet passzolgatni a függvényhívások között. Mindez egyfajta callback-mechanizmust tesz lehetővé, és mindemellett a hívások nyomonkövetésére is alkalmas.

Például:

```
class Command {
public:
    virtual void execute() = 0;
};

class LightUpCommandFake : public Command {
public:
    void execute() { std::cout << "Lampa felkapcsol" << std::endl; }
};
```

```
class LightDownCommandFake : public Command {
public:
    void execute() { std::cout << "Lampa lekapcsol" << std::endl; }
};

class LightUpCommandReal : public Command {
public:
    void execute() { std::cout << "Lampa felkapcsol, nemcsak kiir" <<
        std::endl; }
};

class LightDownCommandReal : public Command {
public:
    void execute() { std::cout << "Lampa lekapcsol, nemcsak kiir" <<
       std::endl; }
};

class LightSwitch {
    Command& upC;
    Command& downC;
public:
    LightSwitch(Command& up, Command& down) : upC(up), downC(down) {}
    void turnUp() { upC.execute(); }
    void turnDown() { downC.execute(); }
};

int main()
{
    LightUpCommandFake lucf; LightDownCommandFake ldcf;
    LightUpCommandReal lucr; LightDownCommandReal ldcr;
    LightSwitch fs(lucf, ldcf); LightSwitch rs(lucr, ldcr);
    fs.turnUp(); fs.turnDown(); rs.turnUp(); rs.turnDown();
}
```

9.4.2. Mediator minta

Probléma: üzeneteket szeretnénk eljuttatni adott típusú objektumokhoz, de mindig azok eltérő csoportjaihoz (részhalmazokat szeretnénk képezni belőlük, és ezen részhalmazoknak elküldeni)

Megoldás: hozzunk létre egy Mediator típust, ami enkapszulálja, hogy mely egyedek tartoznak hozzá, és azoknak juttatja el az üzenetet.

```
template <class T>
class Mediator {
    std::vector<T*> entities;
public:
    void addEntity(T* ep) { entities.push_back(ep); }
```

```cpp
    void distributeMessage(T* sender, const std::string& msg) {
        for (auto elem : entities) {
            if (elem != sender) {
                elem->receiveMsg(sender, msg);
            }
        }
    }
};

class Colleague {
std::string name;
public:
    Colleague(const std::string& nm) : name(nm) {}
    void sendMsg(Mediator<Colleague>* mp, const std::string& msg) {
        mp->distributeMessage(this, msg);
    }
    void receiveMsg(Colleague* cp, const std::string& msg) {
        std::cout << name << " received message from " << cp->name;
        std::cout << std::endl << "\t" << msg << std::endl;
    }
};

int main()
{
    Colleague adam("Szabo Adam");
    Colleague eva("Kuti Eva");
    Colleague utalatos("Hernyo Guszti");

    Mediator<Colleague> baratok;
    baratok.addEntity(&adam);
    baratok.addEntity(&eva);

    Mediator<Colleague> mindenki;
    mindenki.addEntity(&adam);
    mindenki.addEntity(&eva);
    mindenki.addEntity(&utalatos);

    adam.sendMsg(&mindenki, "Talalkozzunk holnap 3-kor");
    eva.sendMsg(&baratok, "OK, de Gusztit meg ne hivd ebedre!");
}
```

9.4.3. Observer minta

Probléma: valamilyen eseményekről értesülnie kell a program más részének, de a kommunikációt nem direktben szeretnénk összehuzalozni.

Megoldás: létrehozhatunk egy Observable interfészt, amin keresztül az Observerek beregisztrálhatják magukat. Ezek után az observerek mindegyike értesülni fog az adott eseményről.

Ez a minta nagyon sok szempontból hasonlít a Mediator-hoz, mivel a Mediator esetében is „beregisztráltunk” valamit, ami aztán üzeneteket kapott. Persze a kettő nem teljesen ugyanaz, mert:

- A Mediator minta esetében maga az entitás kezdeményezte az üzenetküldést, Observer esetében az observable objektum kezdeményezi saját maga
- Az observerek elvben egymástól nagyon különböző típusok is lehetnek, csak az a lényeg, hogy mindegyik megvalósítsa az Observer interfészt (igen, a Mediator esetben is lehetett volna `T` egy absztrakt ősosztály típust létrehozni)

```
// ez az osztaly azert kell, hogy az Observable osztalyban
// el tudjuk tarolni, valamint hogy interfeszt adjon a
   kommunikaciohoz...
class Observer {
public:
    virtual void update(std::string&, double, int) = 0;
};

class Observable { // subject-nek is hivhato
protected:
    std::vector<Observer*> observers;
public:
    void registerObserver(Observer* ob) { observers.push_back(ob); }
    void deregisterObserver(Observer* ob) {
        observers.erase(
            std::remove(
                observers.begin(), observers.end(), ob
            ),
            observers.end()
        );
    }
};

class Environment : public Observable {
    std::string name;
public:
    Environment(const std::string& nm) : name(nm) {}
    void updateInfo(double temp, int humi) {
        for (auto elem : observers) {
            elem->update(name, temp, humi);
        }
    }
};

class WeatherDataObserverEng : public Observer {
```

```cpp
public:
    void update(std::string& observableNm, double temp, int humi) {
        std::cout << "WeatherDataObserver for ";
        std::cout << observableNm << ": temperature is " << temp << "
            deg C - ";
        std::cout << humi << "% humidity" << std::endl;
    }
};

class WeatherDataObserverHun : public Observer {
public:
    void update(std::string& observableNm, double temp, int humi) {
        std::cout << "WeatherDataObserver ";
        std::cout << observableNm << " kornyezetre: a homerseklet "
           << temp << " fok C - ";
        std::cout << humi << "% paratartalom" << std::endl;
    }
};

int main()
{
    Environment e1("nappali"); Environment e2("haloszoba");
       Environment e3("jatszoter");
    Observer* wdoe = new WeatherDataObserverEng;
    Observer* wdoh = new WeatherDataObserverHun;

    e1.registerObserver(wdoe); e2.registerObserver(wdoe);
    e2.registerObserver(wdoh); e3.registerObserver(wdoh);
    e1.updateInfo(22, 50); e2.updateInfo(20.5, 48); e3.updateInfo(33,
        81);
    e2.deregisterObserver(wdoe); e2.updateInfo(20.3, 49);
}
```

9.5. Jellemző kritikák a design patternekkel szemben

Gyakran hallani olyan kritikákat, hogy a design patternek túlságosan limitálják a fejlesztő gondolkodását, és hogy valójában nyelvfüggő is, hogy mi oldható meg elegánsan és kéneyelmesen. E mellett van, aki szerint a design patternek erőltetettek, sok esetben túlzottan bonyolultak amikor egyszerűbb megoldás is jó lehet.

Kétségtelen tény, hogy nem feltétlenül szükséges minden típushoz ősosztályt készíteni, csak azért, hogy legyen. De ami még fontosabb: ezt senki nem is mondta, hogy így kellene csinálnunk! Soha senki nem gondolta, hogy ezeket a mintákat egy-az-egyben át kellene vennünk.

Véleményem szerint a design patternekből sokat lehet tanulni. Teljeskörű megoldást adnak bizonyos problémakörökre, melyeket aztán adaptálhatunk a saját helyzetünkre.

Irodalomjegyzék

[1] Stroustrup Bjarne. *The C++ Programming Language – 4th Edition.* Addison-Wesley, 2013.