C++

13. HÉT – KIVÉTELEK ÉS MONÁDOK (KÉT EGYMÁSTÓL LÁTSZÓLAG TÁVOL ÁLLÓ TÉMA)

Milyen hibák léphetnek fel futás közben?

Általánosságban véve elkerülhetetlen, hogy egy kódot futtatva hibák lépjenek fel

- A fordító statikus elemzőként nem tud minden hibát megtalálni (típus-szűkítések, és még nagyon sok minden áteshet a rostán)
- Előfordulhatnak logikai hibák, melyekre csak futáskor derül fény
  - Pl. tomb túlcízmzése, egy API nem megfelelő használata (szemantikai értelemben).
- Előfordulhatnak várt vagy nem várt futásidejű hibaesemények is
  - Pl. nincs lefoglalható megmória, nem érhető el a hálózat, …
  - Ez utóbbiakre elvben felkészülhetünk, de konkrétan nem tudjuk, melyik futáskor fognak fellépni

Klasszikus megoldás: hibakódok

A klasszikus imperatív modellben gyakran dolgoztak hibakódokkal

- A függvények pl. egy bool vagy int értékkel tértek vissza, melyek értéke arra utalt, hogy volt-e hiba, vagy milyen hiba lépett fel (int esetében)
- Ilyenkor, ha a függvényeknek más visszatérési értéke is Kellett, hogy legyen, azt egy bemenő referenciával vagy pointerrel lehetett kívülről elérni (output argumentum)

Napjainkban ez a megközelítés ritka, de léteznek alacsonyszintű API-k, ahol még divatos

A módszer hátulütője, hogy szinte több kód foglalkozik a hiba fennállásának ellenőrzésével és kezelésével, mint magával az üzleti logikával

Kivételek (exceptions)

A kivételeket pont azért vezették be sok nyelven, hogy ne kelljen minden függvényhívásnál hibakódot ellenőrizni

- Máskülönben a kód úgy nézett ki, hogy 1 függvény-hívásra 4-5 sor “overhead” jut

A kivétel felfogható úgy, mint egy objektum, ami:

- Hiba fellépésekor jön létre (mi is létrehozhatunk explicite, vagy egy általunk használt API kódja hozza létre)
- Automatikusan “tovább terjesztődik” (a program “kivételt dob”) a hibakezelés helye felé, ahol végül a kivételt “elkapjuk” (vagy ha nem kapjuk el, akkor crash-el a program)
- A kivételkezelés helye lehet valami egészen más részében a kódnak, mint ahol a hibás függvényhívás történik

Kivételek (exceptions)

A kivételkezelés helye lehet:

- A hibára futó F függvény meghívását körbevevő try-catch szerkezet catch blokkja, ha ez a catch blokk alkalmas az adott kivétel-típus elkapására
- A hibára futó függvény hívását tartalmazó F függvényt meghívó F1 függvényben levő try-catch szerkezet catch blokkja, ha a szerkezet try blokkjában benne van az F függvény meghívása és a catch blokk alkalmas az adott kivétel-típus elkapására
- Az F1 függvényt meghívó F2 függvényben levő try-catch szerkezet catch blokkja, ha a szerkezet try blokkjában benne van az F1 függvény meghívása és a catch blokk alkalmas az adott kivétel-típus elkapására… és így tovább

Mivel ilyenkor a futtatási környezetnek vissza kell térnie a függvényhívási láncolatban egyészen addig a függvényig, ahol a programozó catch ágat definiált a kivátel elkapására, ezt a működést úgy his hívják, hogy “stack unwinding”

- A stacken levő hívás-láncolat “visszatekeredik”, mint egy cérnagombolyag.

Kivételek (exceptions)

A kivételek legfőbb előnyei:

- Az API-t meghívó kódot rákényszerítik, hogy gondolkozzon el, helyesen használja-e az API-t és hogy milyen váratlan események léphetnek fel
- Az API kódot rákonyszeríti, hogy tegye világossá, milyen feltételekkel biztosítja a működését, és mikor dob kivételt
  - Ha az API kódja olyan kivételt dob, ami nincs előre lefektetve a dokumentációban, az API kódja hibás
  - Ha az API olyan kivételt dob, ami előre le lett fektetve a dokumentációban, de a kivételt nem kapja el semelyik függvény, akkor az API-t hívó kód hibás
  - -> éppen ezért szokás az API-khoz saját kivétel típusokat definiálni, hogy már a névből megállapítható legyen a különbség
- Mivel a stack unwinding mechanizmus automatikus, nem kell azon gondolkodni, hogy a hívást követően az eredmény használható-e (ha nem, automatikusan eldobódik a kivétel). Így a kivétel fellépésének helye és lekezelésének a helye jól elszeparált
- A kivétel fellépése és kezelése közötti kódban létrejött változók felszabadítása a scope-nak megfelelően történik (de az inkonzisztens állapotokra figyelni kell abban az esetben, ha egy változó ezt túléli)

Kivételek (exceptions)

A kivételek legfőbb hátrányai (legalábbis a funkcionális programozás “felkent papjai” szerint):

- Ha egy függvény kivételt dob, már nem lehet tiszta (pure), mivel ugyanarra a bemenetre két különböző választ is adhat
  - Például ha az egyik esetben elérhető a hálózat és van memória, a másikban nem
- Mi történik, ha tolerálni szeretnénk, hogy egy függvény 1 esetben hibára futott, de 99 esetben nem?
  - Pl. ha egy konténer elemein futtatunk végig egy adott függvényt (map), de az egyik hívás sikertelen, attól még a többi eredményt nem biztos, hogy el kell dobnunk
- Majd látni fogjuk, hogy a funkcionális programozási paradigma sokkal megengedőbb a futásidejű hibákkal szemben (persze azokat ugyanúgy kezelni kell, de nem szeretik megakasztani ezért a teljes információ-feldolgozási láncolatot). Többek között ezért fejlesztettek ki olyan adatszerkezeteket, mint a monadikus típusokat, melyek a hibák és egyéb mellékhatások fellépését mintegy összetett objektumként modellezik

Kivételek dobása C++-ban

A kivétel a C++-ban olyan objektum, mint bármelyik másik

Ideális esetben az objektum típusa származik egy kivétel ősosztályból, de ez nem követelmény. Egy stringet, vagy int-et is használhatunk kivételként, ha a throw utasítással eldobjuk

Ehhez mindössze annyi szükséges, hogy a throw kulcsszóval a hiba fellépésekor eldobjuk a kivételt.

- A kivételt elkapni egy try-catch blokkal lehetséges, feltéve, hogy a kivétel egy olyan kódrészből kerül eldobásra, amely a try blokk részeként kerül (közvetlenül vagy közvetve) meghívásra
- A catch blokkban megadható, hogy milyen kivétel típusokra vagyunk receptívek. Több catch blokk is készíthető egyszerre, ld. Példa:

Példa: throw, try, catch

Itt a kivételt nem sikerült elkapnunk, mert a catch blokkban csak int-ekre figyelünk:

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


void f(int input) {
    if (input == 1) {
        throw "String Exception";
    }
    if (input == 2) {
        throw 55;
    }
    std::cout << "no exception was thrown" << std::endl;
}

int main() {
    try{
        f(3);
        f(2);
        f(1);
    } catch (int e) {
        std::cout << "Exception: " << e << std::endl;
    }

    try {
        f(1);
    } catch (int e) {
        std::cout << "Exception: " << e << std::endl;
    }
}
```

```
no exception was thrown
Exception: 55
terminate called after throwing an instance of 'char const*'
bash: line 7: 31941 Aborted                 (core dumped) ./a.out
```

Példa: throw, try, catch

Így már jók vagyunk. Láthatjuk azt is, hogy a nagyobb objektumokat referenciával is elkaphatjuk:

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


void f(int input) {
    if (input == 1) {
        throw std::string("String Exception");
    }
    if (input == 2) {
        throw 55;
    }
    std::cout << "no exception was thrown" << std::endl;
}

int main() {
    try{
        f(3);
        f(2);
        f(1);
    } catch (int e) {
        std::cout << "Exception: " << e << std::endl;
    }

    try {
        f(1);
    } catch (int e) {
        std::cout << "Exception: " << e << std::endl;
    } catch (std::string& s) {
        std::cout << s << std::endl;
    }
}
```

```
no exception was thrown
Exception: 55
String Exception
```

Kivételek dobása érték és elkapása referencia szerint

A throw kulcsszó bármilyen objektumot alapértelmezett esetben érték szerint továbbít. Érdemes ezeket a catch blokkban referenciaként elkapni (hacsak nem pointerről van szó, mint const char* vagy egyéb).

Ennek az az előnye, hogy futáskor a fedél alatt nem kell lemásolni az objektumot, plusz dinamikus polimorfizmus is használható!

- Ha egy konkrét típusú kivételt dobunk, elkapható annak egy szülőosztályra mint típusra hivatkozó referenciával is
- Ha a kivétel értékét nem módosítjuk, elkaphatjuk const referenciaként is

std::exception

Általában nem bármilyen, hanem az std::exception osztályból származtatott (vagy azokból tovább származtatott saját) típust célszerű használnunk. A sztenderd könyvtár által dobott kivételek mind ilyenek:

- logic_error
  - invalid_argument
  - domain_error
  - length_error
  - out_of_range
  - future_error(C++11)
- bad_optional_access(C++17)

- runtime_error
  - range_error
  - overflow_error
  - underflow_error
  - regex_error(C++11)
  - system_error(C++11)
    - ios_base::failure(C++11)
    - filesystem::filesystem_error(C++17)
  - tx_exception(TM TS)
  - nonexistent_local_time(C++20)
  - ambiguous_local_time(C++20)
  - format_error(C++20)

- bad_typeid
- bad_cast
  - bad_any_cast(C++17)
- bad_weak_ptr(C++11)
- bad_function_call(C++11)
- bad_alloc
  - bad_array_new_length(C++11)
- bad_exception
- ios_base::failure(until C++11)
- bad_variant_access(C++17)

std::exception

Az std::exception osztály tartalmazza az alábbi virtuális metódust:

```
virtual const char* what() const noexcept;
```

Ennek override-olásával tudjuk elérni – amikor kiterjesztjük az osztályt – hogy a felhasználó a catch blokkban információt tudjon kérni az elkapott hibáról

noexcept

A C++11 óta a noexcept kulcsszóval jellemzett függvények / metódusok esetén a fordító kikényszeríti, hogy az adott függvény / metódus ne dobjon kivételt. Interfészek deklarálásánál ez hasznos információ tud lenni más programozók számára, ahogy pl. a move konstruktor és assignment esetén szinte kötelező is használni, mert akkor a fordító tudhatja, hogy biztonságosan mozgathat

```
5  void f(int input) noexcept {
6      if (input == 1) {
7          throw std::string("String Exception");
8      }
9      if (input == 2) {
10         throw 55;
11     }
12     std::cout << "no exception was thrown" << std::endl;
13 }
14
15 int main() {
16     try {
17         f(1);
18     } catch (int e) {
19         std::cout << "Exception: " << e << std::endl;
20     } catch (std::string& s) {
21         std::cout << s << std::endl;
22     }
23 }
```

```
main.cpp: In function 'void f(int)':
main.cpp:7:9: warning: 'throw' will always call 'terminate' [-Wterminate]
    7 |         throw std::string("String Exception");
      |         ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
main.cpp:10:9: warning: 'throw' will always call 'terminate' [-Wterminate]
   10 |         throw 55;
      |         ^~~~~~~~
terminate called after throwing an instance of 'std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> >'
bash: line 7:  6897 Aborted                 (core dumped) ./a.out
```

Functorok és monádok

A functorok és monádok olyan kategóriaelméleti fogalmak, melyek többek között a kivételkezelésre (de rengeteg más problémára is!) is elegáns alternatívát nyújtanak.

A kategóriaelmélet a múlt század közepe táján alakult ki abból a célból, hogy látszólag különböző, de szerkezetükben azonos problémákat matematikailag egységesen lehessen kezelni.

- Ha egy halmazról és az elemein elvégezhető műveletekről tudom, hogy egy adott kategóriába tartoznak, érteni fogom, hogy nagyjából milyen tulajdonságokat várhatok el tőle, és hogyan tudom őket hatékonyan leprogramozni
- Esetünkben nem kell túlzottan mélyen belemenni a matematikába, szerencsére. Az alapelveket és használatot e nélkül is meg lehet érteni!

Functorok

(Vigyázat! A functor szót itt kategóriaelméleti értelemben használjuk, nem úgy, mint korábban, amikor függvény objektumot értettünk alatta! – e téren a szakirodalom sem egységes)

Functornak nevezünk minden olyan template-elt osztályt, melynek 1 template típusa van (tehát mint Functor<T>), és amelyre definiálva van egy *map(f, a)* meghívható egyed, melyre teljesül, hogy:

- map(f, a) akkor és csak akkor legális kifejezés, ha:
  - a egy Functor<T> típusú argumentum
  - f egy olyan meghívható egyed, ami T típusú bemenetet vár és egy U típusú kimenetet ad
  - map(f,a) kimenetének típusa Functor<U>
- Ha f(x) = x, akkor map(f, a) kimenete *a* lesz
- map(g, map(f, a)) eredménye pontosan ugyanaz, mint map(gf, a) eredménye, ahol “gf” g és f kompozícióját jelenti, amely adott bemenetre először f függvényt, majd az eredményre g függvényt hajtja végre

Functorok

Intuitív értelmezés: **a functor egyfajta wrapper típus**, mely a *map()*-en keresztül lehetővé teszi, hogy az általa becsomagolt típuson műveleteket hajthassunk végre anélkül, hogy ki kelljen csomagolni

- Pontosabban: a map kicsomagol, végrehajt, és visszacsomagol

Az, hogy adott functor hány “belső elemet” tartalmaz, teljesen mindegy. Egy *std::vector* vagy *std::list* is lehet functor, ha a *map()* végigmegy minden elemen, és a meghívott függvény eredményeit ismét egy std::vector vagy std::list konténerbe csomagolja

De ha úgy vesszük, *std::optional* is lehet egy functor. Egy ilyen típus opcionálisan egyetlen értéket tartalmaz. Ha van benne érték, arra lefuttatható a függvény, az eredmény pedig visszapakolható egy optional-ba. Ha nincs benne érték, akkor az üres optional-t adhatja vissza a *map()* is.

Functorok

```cpp
#include <string>
#include <iostream>
#include <optional>
#include <functional>


template <typename T, typename U>
std::optional<U> map(const std::function<U(T)>& f, const std::optional<T>& a) {
    if (a) {
        return std::optional<U>(f(*a));
    }
    return std::nullopt;
}
```

```cpp
int main() {
    auto plus1 = [](int x){return x + 1;};
    std::optional<int> one{1};
    std::optional<int> none{std::nullopt};

    std::optional<int> res1 = map<int, int>(plus1, one);
    std::optional<int> res2 = map<int, int>(plus1, none);
    if (res1) {
        std::cout << "res1 is " << *res1 << std::endl;
    }
    if (!res2) {
        std::cout << "res2 is empty" << std::endl;
    }
}
```

```
res1 is 2
res2 is empty
```

Functorok

A functorok hibakezelés (és egyéb mellékhatások kezelése) szempontjából is pont ezért hasznosak.

Ha van egy f függvényem, ami A típusból B típust csinál, de néha elhasalhat, a hibakezelést “elodázhatom”, ha a konkrét A típusú változóimat egy functorba csomagolom, és egy alkalmas map-pel hívom meg rá az f függvényt.

- A működés lényege ekkor, hogy a functor vagy használható értéket, vagy egy hibajegyet tartalmaz, és a rá meghívott map vagy tovább viszi a számításokat, vagy csendben nem csinál semmit. Ezután, ha minden művelettel végeztem, megvizsgálhatom a végezetül kapott functor objektumot, hogy van-e benne értelmes eredmény)

Functorok

Messzebbről nézve a functor és a rá definiált *map()* függvény alkalmas lehet arra is, hogy egy eredeti funkcionalitást “kidekoráljunk” valami járulékos művelettel.

- Pl. lehetséges olyan, hogy a map() végrehajtja az adott függvényt a becsomagolt változón, de közben logolja is, hogy meghívtuk a függvényt vagy hogy milyen köztes eredményt kaptunk.

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

template <typename X>
class MapLogger {
    X wrappedVal;
public:
    MapLogger(X x) : wrappedVal(x) {}
    const X& getVal() { return wrappedVal; }
};
```

```
template <typename X, typename U>
MapLogger<U> map(std::function<U(X)> f, MapLogger<X> inp) {
    U result = f(inp.getVal());
    std::cout << "intermediate result: " << result <<
std::endl;
    return MapLogger<U>(result);
}
```

Functorok

```
int main()
{
    int x = 0;
    auto incr = [](int inp) -> int {return inp + 1; };
    auto dupl = [](int inp) -> int {return inp * 2; };
    std::cout << "One way to do this: " <<
dupl(incr(incr(x))) << std::endl;

    MapLogger<int> xml(0);
    map<int, int>(dupl,
        map<int, int>(incr,
            map<int, int>(incr, xml)
        )
    );
}
```

```
One way to do this: 4
intermediate result: 1
intermediate result: 2
intermediate result: 4
```

Functorok

Ez érdekes példa, de egy lehetséges kritikai észrevétel, hogy ebben a formában még egy kicsit kényszeredett.

- Pl. azért, mert a map(f, functor<X>) függvény nem tudhat sokat arról, hogy milyen f()-et adtunk át neki, vagy hogy mik a Functor<X> típus tulajdonságai… ugyanúgy, ahogy a Functor<X> típus sem tudhatja, hogy milyen dolgokat csinálunk az általa becsomagolt X típusú értékkel

Hasznosabb lenne, ha ezt az egészet egy olyan formában tudnánk megfogalmazni, ahol a végrehajtandó f függvény az X alaptípusból nem egy U alaptípust, hanem egy M<U> csomagolt típust hoz létre.

- Ekkor maga f gondoskodna arról, hogy a csomagoló típusú változónak (M) átadja az f-re jellemző információkat, vagyis ezáltal rugalmasabb kódot kapnánk!
- Vegyük észre, hogy ha ez működne is, a functor-map világban nem lenne életképes, mert az M<U> típusú eredményt a *map()* is becsomagolná a saját functorába – innentől egy M<M<U>> típust kapnánk és erre a *map()* már nem lenne alkalmazható (legalábbis nem olyan függvénnyel, ami Y típusú bemenetet vár)

Monadok

Tulajdonképpen erre a problémára (is) születtek a monádok. Vagyis pontosabban: a monádok ezt a dilemmát (is) képesek feloldani.

A funkcionális programozásban egy **M<A> típusú változót monádnak nevezünk**, ha M<A> egy functor is egyben (tehát létezik hozzá egy már ismert formátumú *map(f, m<A>)* függvény), de még létezik hozzá további 2 másik függvény is:

- *pure(a)* – vagy más néven *unit(a)*, amely egy A típusú a-ból M<A>-t csinál (ez olyan, mint egy konstruktor)
- *join(M<M<A>>),* amely egy M<M<A>> típusból csinál egy M<A> típust – magyarán kicsomagolja a monád belsejét, ha az is egy monád
  - A *join()*-ra vonatkozó egyetlen formális feltétel, hogy ha többszintű a beágyazás, akkor a joinolás sorrendje mindegy
- Sok esetben a *map()*-et és a *join()*-t egyetlen függvényben valósítják meg, amit *bind()*-nak vagy *flatmap()*-nek szoktak nevezni.

Monadok

Ha jól belegondolunk, a *bind() / flatmap()* függvénynek az lesz az értelme, hogy olyan függvényt is lehessen *map()*-elni, amely A típusból nem B, hanem M<B> típust csinál.

Az alap *map()* ilyenkor egy M<A> bemenetből M<M<B>>-t csinálna, viszont ha ezután rögtön hívunk rá egy *join()*-t, visszakapjuk a kívánt M<B> típust.

Tegyük fel például, hogy van egy vektorunk intekkel, és az a feladat, hogy mindegyik intet 3x ismételjük meg.

- Erre az imperatív megoldás az lenne, hogy egy loopban végig megyünk a vektor elemein, és egy új vektorba mindegyiket 3x belepakolnánk
- A funkcionális megoldás viszont az lenne, hogy mappelnénk az eredeti vektorra egy olyan függvényt, amelyik mindegyik eleméből 3-at ad vissza egy konténerben, pl. vektorban. De ha ezeket az eredményeket belepakolnánk egy vektorba, akkor vektorokból álló vektort kapnánk! A *join(map())* – vagy más néven *bind() / flatmap()* viszont képes arra, hogy ezt kilapítsa.

Monadok

Hogyan nézne ki az előző példa monádokkal?

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

template <typename X>
class LoggerMonad {
    X wrappedVal;
    std::string msgToLog;
public:
    LoggerMonad(X x, const std::string& stl) :
wrappedVal(x), msgToLog(stl) {}
    const X& getVal() { return wrappedVal; }
    const std::string& getMsg() { return msgToLog; }
};
```

```
template <typename X, typename U>
LoggerMonad<U> flatmap(std::function<LoggerMonad<U>(X)> f,
LoggerMonad<X> inp) {
    LoggerMonad<U> result = f(inp.getVal());
    std::cout << result.getMsg() << std::endl;
    return result;
}
```

Vegyük észre: a functoros példában a map() függvény hívta meg a MapLogger osztály konstruktorát, mert az ő dolga volt az eredményt visszacsomagolni. Itt viszont f() hívja meg LoggerMonad konstruktorát. De mivel f() többet tud magáról, mint a map() tudott f()-ről, ezért ebben a példában az f() függvény valami “egyedi” dolgot is átadhat a konstruktornak!

Monadok

Hogyan nézne ki az előző példa monádokkal?

```
int main()
{
    auto incr = [](int inp) -> LoggerMonad<int> {
        return LoggerMonad<int>(inp + 1, "called increment to get " +
std::to_string(inp + 1));
    };
    auto dupl = [](int inp) -> LoggerMonad<int> {
        return LoggerMonad<int>(inp * 2, "called duplicate to get " +
std::to_string(inp * 2));
    };

    LoggerMonad<int> xm(0, "");
    flatmap<int, int>(dupl,
        flatmap<int, int>(incr,
            flatmap<int, int>(incr, xm)
        )
    );
}
```

```
called increment to get 1
called increment to get 2
called duplicate to get 4
```

Vegyük észre: a functoros példában a map() függvény hívta meg a MapLogger osztály konstruktorát, mert az ő dolga volt az eredményt visszacsomagolni. Itt viszont f() hívja meg LoggerMonad konstruktorát. De mivel f() többet tud magáról, mint a map() tudott f()-ről, ezért ebben a példában az f() függvény valami “egyedi” dolgot is átadhat a konstruktornak!

Monadok

A main függvény szebb is lehet, ha *flatmap()*-et tagfüggvényként valósítjuk meg:

```
template <typename X>
class LoggerMonad {
    X wrappedVal;
    std::string msgToLog;
public:
    LoggerMonad(X x, const std::string& stl) : wrappedVal(x),
msgToLog(stl) {}
    const std::string& getMsg() { return msgToLog; }
    template <typename U>
    LoggerMonad<U> flatmap(std::function<LoggerMonad<U>(X)> f) {
        LoggerMonad<U> result = f(wrappedVal);
        std::cout << result.getMsg() << std::endl;
        return result;
    }
};
```

Monadok

Így már láncolva is működnek a hívások:

```
int main()
{
    auto incr = [](int inp) -> LoggerMonad<int> {
        return LoggerMonad<int>(inp + 1, "called increment to get " + std::to_string(inp + 1));
    };
    auto dupl = [](int inp) -> LoggerMonad<int> {
        return LoggerMonad<int>(inp * 2, "called duplicate to get " + std::to_string(inp * 2));
    };

    LoggerMonad<int> xm(0, "");
    xm.flatmap<int>(incr).flatmap<int>(incr).flatmap<int>(dupl);
}
```

Monadok

A monádok más esetekben is hatékony megoldást jelenthetnek – pl. vegyük a rekurzió és a tail-call optimization témáját.

Ha pl. egy faktoriális függvényt szeretnénk írni, ezt naív implementációval megtehetnénk így:

```
int factorial(int input) {
    if (input < 2) {
        return 1;
    }
    return input * factorial(input - 1);
}
```

Monadok

A megoldás hátulütője, hogy pl. input = 10.000-nél már stack overflow-val crash-el a program.

Ez azért van, mert factorial(10000) nem tér vissza addig, ameddig factorial(9999) sem… az pedig nem tér vissza, ameddig factorial(9998) vissza nem tér és így tovább. Ez probléma, mert mindegyik függvény kontextusát a stack-en kell tárolni, így a stack csak nő és nő.

Bizonyos nyelvek / fordítók képesek az ún. Tail-Call Optimization (TCO) megvalósítására, ami akkor lehetséges, ha a legutolsó művelet az egyetlen, ami rekurzív.

Ilyenkor a függvény átalakítható olyanra, mintha vissza is térne ilyenkor, az aktuális részeredményt pedig valahol “megjegyezzük”

```
int factorial(int input) {
    if (input < 2) {
        return 1;
    }
    return input * factorial(input - 1);
}
```

Monadok

Pl: mivel tudjuk, hogy factorial(10000) = 10000 * 9999 * factorial(9998), és a szorzás sorrendje mindegy, ezért már az elején összeszorozható az első 2 tag, és már “csak” factorial(9998) kiszámítása marad -> ezért igazából felesleges a kontextust megtartani a stack-en.

Mindenesetre a C++ fordítók nem valósítanak meg Tail-Call Optimization-t, ezért nekünk kell ilyen esetekre valami okosságot kitalálnunk.

Egy lehetséges megoldás az ún. Trambulin (trampoline) pattern, vagy más néven Free Monad.

Monadok

A Free / Trampoline monad egy olyan típus, ami vagy egy visszaadandó értéket, vagy egy folytatandó számítást enkapszulál.

Gondolkozzunk el a faktoriális példa alapján, hogy mire lenne itt szükségünk:

- Ha a bemenet 1 (vagy kevesebb, mint 2), akkor az eltárolt részeredmény visszaadható
- Ha a bemenet nagyobb, mint 1, akkor egy számítás reprezentációját kell visszaadni

Mielőtt egy általános megoldásban gondolkodnánk, vegyük sorra, mit kell eltárolnia a Faktoriális Trambulinnak

- El kell tárolni egy aktuális számlálót (int counter), hogy minek a faktoriálisát akarjuk éppen kiszámítani
- El kell tárolni egy aktuális részeredményt (ez is alapesetben int, de jó ha BigInt típust használunk, mert nagyon nagy értékek tudnak lenni, amik int-en már nem is reprezentálhatóak – itt egy példa BigInt könyvtár, ami egyetlen .h fájlból áll: https://github.com/faheel/BigInt
- El kell tárolni egy doBounce nevű bool típust, ami megmondja, hogy végeztünk-e a számítással vagy folytatni kell

Monadok

A megoldás első iterációban:

```
class FactorialTrampoline {
    BigInt currentRes;
    int counter;
    bool doBounce;
public:
    FactorialTrampoline(int initialCounter) : counter(initialCounter), currentRes(1), doBounce(true) {
        if (initialCounter < 2) doBounce = false;
    }

    FactorialTrampoline(int initialCounter, BigInt initialResult, bool doBounce) :
        counter(initialCounter), currentRes(initialResult), doBounce(doBounce) {}

    BigInt getResult() { return currentRes; }
```

Monadok

A kétféle konstruktor azért jó, mert alapesetben elég az egyszerűbb, de ha a flatmap()-nek átadott függvény további trambulinokat gyárt, meg kellhet adni minden infot. A folytatás:

```
    static FactorialTrampoline embeddedOperation(int counter, BigInt currentRes) {
        if (counter < 2) { return FactorialTrampoline(counter, currentRes, false); }
        else { return FactorialTrampoline(counter - 1, currentRes * counter, true); }
    }

    FactorialTrampoline flatmap() { // nincs arg mert van a static embeddedOperation
        FactorialTrampoline tmp(counter, currentRes, doBounce);
        while (true) {
            if (!tmp.doBounce) return tmp;
            tmp = FactorialTrampoline(FactorialTrampoline::embeddedOperation(tmp.counter, tmp.currentRes));
            if (tmp.counter % 500 == 0) std::cout << "\t currently at: " << tmp.counter << std::endl;
        }
    }
};
```

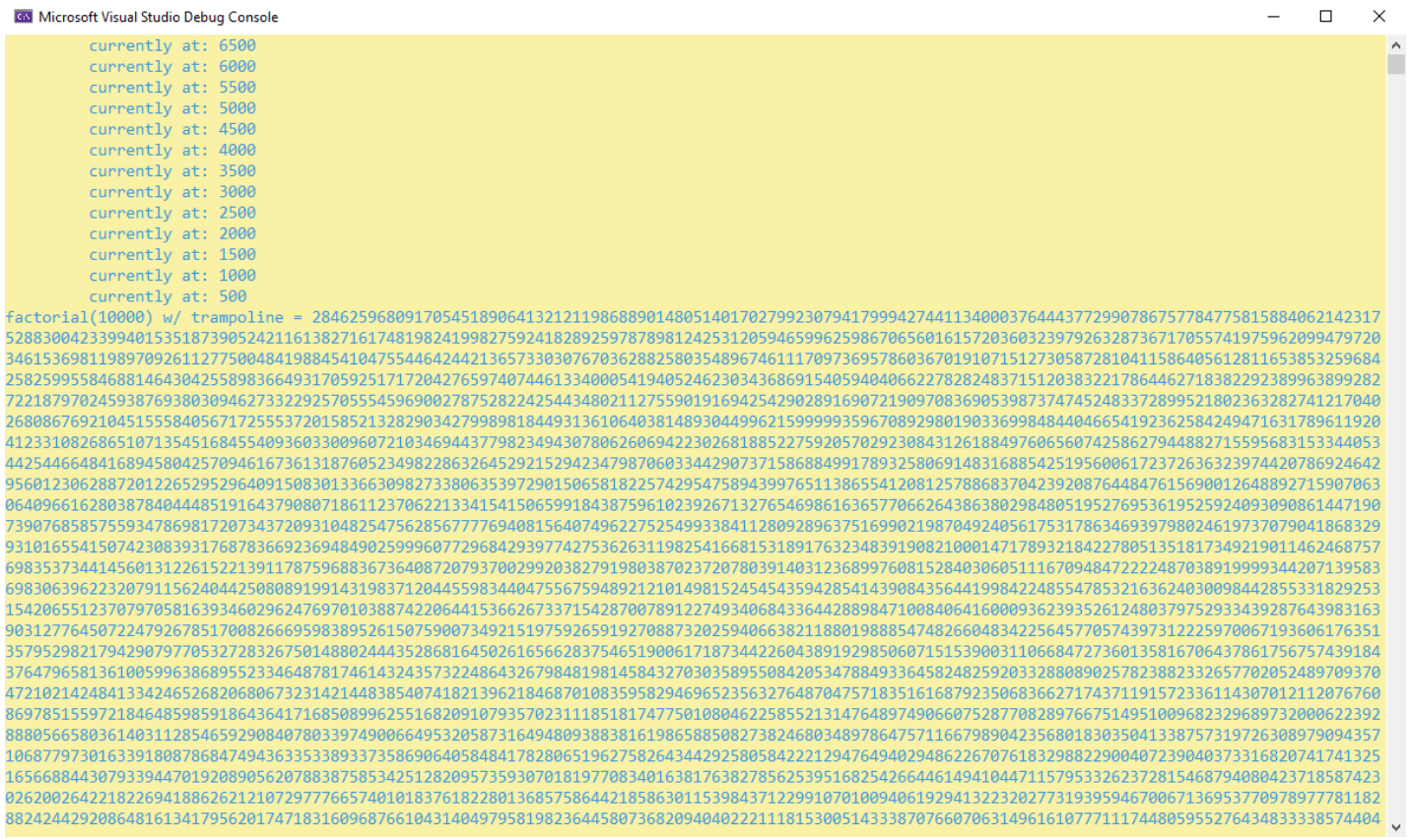
Monadok

Világos, hogy itt jónéhány dologban eltértünk az előírásoktól – nincs *pure()* vagy *unit()* függvény, *flatmap()* igazából nem vár függvényt mert van egy static műveletünk, amit benne felhasználunk. És valójában nincsen template típusunk amit becsomagolnánk, csak bedrótozott alaptípusaink vannak.

- De az alapvető minta mégis az, hogy az *embeddedOperation* függvény az alaptípusokból egy ugyanolyan típust csinál, mint amit monádként létrehoztunk. Tehát a rugalmasság elvi lehetősége megvan.
- Az eredményt jó sokáig tart kiszámolni, de sikerül, mindenféle stack overflow nélkül:

```
int main()
{
    std::cout << "factorial(10) = " << factorial(10) << std::endl;
    FactorialTrampoline fact10000(10000);
    std::cout << "factorial(10000) w/ trampoline = " << fact10000.flatmap().getResult() << std::endl;
    //std::cout << "factorial(10000) = " << factorial(10000) << std::endl;
}
```

Monadok


Microsoft Visual Studio Debug Console
currently at: 6500
currently at: 6000
currently at: 5500
currently at: 5000
currently at: 4500
currently at: 4000
currently at: 3500
currently at: 3000
currently at: 2500
currently at: 2000
currently at: 1500
currently at: 1000
currently at: 500
factorial(10000) w/ trampoline = 2846259680917054518906413212119868890148051401702799230794179994274411340003764443772990786757784775815884062142317
5288300423399401535187390524211613827161748198241998275924182892597878981242531205946599625986706560161572036032397926328736717055741975962099479720
3461536981198970926112775004841988454104755446424421365733030767036288258035489674611170973695786036701910715127305872810411586405612811653853259684
2582599558468814643042558983664931705925171720427659740744613340005419405246230343686915405940406622782824837151203832217864462718382292389963899282
7221879702459387693803094627332292570555459690027875282242544348021127559019169425429028916907219097083690539873747452483372899521802363282741217040
2680867692104515558405671725553720158521328290342799898184493136106403814893044996215999993596708929801903369984844046654192362584249471631789611920
4123310826865107135451684554093603300960721034694437798234943078062606942230268188522759205702923084312618849760656074258627944882715595683153344053
4425446648416894580425709461673613187605234982286326452921529423479870603344290737158688499178932580691483168854251956006172372636323974420786924642
9560123062887201226529529640915083013366309827338063539729015065818225742954758943997651138655412081257886837042392087644847615690012648892715907063
0640966162803878404448519164379080718611237062213341541506599184387596102392671327654698616365770662643863802984805195276953619525924093090861447190
7390768585755934786981720734372093104825475628567777694081564074962275254993384112809289637516990219870492405617531786346939798024619737079041868329
9310165541507423083931768783669236948490259996077296842939774275362631198254166815318917632348391908210001471789321842278051351817349219011462468757
6983537344145601312261522139117875968836736408720793700299203827919803870237207803914031236899760815284030605111670948472222487038919999344207139583
6983063962232079115624044250808919914319837120445598344047556759489212101498152454543594285414390843564419984224855478532163624030098442855331829253
1542065512370797058163934602962476970103887422064415366267337154287007891227493406843364428898471008406416000936239352612480379752933439287643983163
9031277645072247926785170082666959838952615075900734921519759265919270887320259406638211880198885474826604834225645770574397312225970067193606176351
3579529821794290797705327283267501488024443528681645026165662837546519006171873442260438919298506071515390031106684727360135816706437861756757439184
3764796581361005996368695523346487817461432435732248643267984819814584327030358955084205347884933645824825920332880902578238823326577020524897093704
7210214248413342465268206806732314214483854074182139621846870108359582946965235632764870475718351616879235068366271743711915723361143070121120767606
8697851559721846485985918643641716850899625516820910793570231118518174775010804622585521314764897490660752877082897667514951009682329689732000622392
8880566580361403112854659290840780339749006649532058731649480938838161986588508273824680348978647571166798904235680183035041338757319726308979094357
1068779730163391808786847494363353389337358690640584841782806519627582643442925805842221294764940294862670761832988229004072390403733168207417413251
6566884430793394470192089056207883875853425128209573593070181977083401638176382785625395168254266446149410447115795332623728154687940804237185874230
0262002642218226941886262121072977766574010183761822801368575864421858630115398437122991070100940619294132232027731939594670067136953770978977781182
8824244292086481613417956201747183160968766104314049795819823644580736820940402221118153005143338707660706314961610777111744805955276434833338574404

Monadok

Általánosítsuk most ezt egy monádabb monádra, és alkalmazzuk a Fibonacci számok kiszámítására!

Vegyük észre, hogy a Fibonacci számokat meghatározó naív függvény önmagában sem optimalizálható TCO-val, mivel nem 1, hanem 2 rekurzív hívás is van benne. A naív implementáció:

```
int fibonacci(int number) {
    if (number < 3) {
        return 1;
    }
    return fibonacci(number - 1) + fibonacci(number - 2);
}
```

Monadok

A Fibonacci függvénynél általában meg szokták említeni, hogy azért is nagyon pazarló, mert sok mindent többször kell kiszámolni. Például *fibonacci(5)*-öt akkor is kiszámoljuk, amikor a cél *fibonacci(6)* kiszámítása, és akkor is, amikor a cél *fibonacci(7)* kiszámítása, amik pedig minden további számhoz többször is kellenek.

Egyfajta megoldás erre a *memoizáció*, amikor meglévő részeredményeket külön el is tárolulnk, hogy bizonyos rekurzív ágak levághatóak legyenek.

Ettől a stack overflow elkerülése részben független kérdés persze.

Monadok

Nézzünk egy lehetséges megoldást! Maga az általános trambulin (a copy assignment az utolsó tmp = operation... sor miatt szükséges):

```
template <typename X>
class Trampoline {
    X& state;
    bool doBounce;
public:
    Trampoline(X& state) : state(state), doBounce(true) {}
    Trampoline(X& state, bool dbnce) : state(state), doBounce(dbnce) {}
    Trampoline<X>& operator=(const Trampoline<X>& other) {
        state = other.state; doBounce = other.doBounce;
        return *this;
    }
    Trampoline<X> flatmap(std::function<Trampoline<X>(X&)> operation) {
        Trampoline<X> tmp(state, doBounce);
        while (true) {
            if (!tmp.doBounce) { return tmp; }
            tmp = operation(state);
        }
    }
};
```

Monadok

Maga a state rajtunk múlik, hogyan alakítjuk ki. Az egyszerűség kedvéért structot használunk, minden publikus hogy ne kelljen getterekkel / setterekkel bajlódni:

A cache a memoizációt szolgálja

currentCounters a még feldolgozandó fibonacci számokat tartalmazza

currentRes az aktuális összeg, BigInt formátumban

```
struct FibonacciState {
    std::map<int, BigInt> cache;
    std::deque<int> currentCounters;
    BigInt currentRes;
    FibonacciState(int currentCounter, BigInt currentRes) :
currentRes(currentRes) {
        currentCounters.push_front(currentCounter);
    }
};
```

Monadok

A *flatmap()* szerepe pedig annyi, hogy az általa lefuttatott művelet vesz egy state-et, feldolgozza a benne levő első (n értékű) countert (ha szükséges, az n-1 és n-2 höz tartozó countert is beleteszi), vagy frissíti currentRes-t és visszaad egy új trambulint.

Nézzük magát az operátort első megközelítésben:

Monadok

Az operator első fele:

```
Trampoline<FibonacciState> fibonacciWorkhorse(FibonacciState& fibStateRef) {
    if (fibStateRef.currentCounters.size() == 0) { // ha nincs mit csinalni...
        return Trampoline<FibonacciState>(fibStateRef, false); // vegeztunk
    }

    int cntr = fibStateRef.currentCounters[0]; // fel kell dolgozni fibonacci(x)-et, ahol x kisebb mint az eredeti arg
    fibStateRef.currentCounters.pop_front(); // ki is szedjuk currentCounters-bol

    if (fibStateRef.cache.find(cntr) != fibStateRef.cache.end()) { // ha egyszer mar kiszamoltuk...
        fibStateRef.currentRes += fibStateRef.cache[cntr]; // csak hozzaadjuk az erteket
        return Trampoline<FibonacciState>(fibStateRef, true);
    }
    else { // ha meg nem szamoltuk ki akkor az a kerdes, hogy tudjuk-e az erteket
```

Monadok

Az operator második fele:

```
        else { // ha meg nem szamoltuk ki akkor az a kerdes, hogy tudjuk-e az erteket
            if (cntr < 3) {
                fibStateRef.cache[cntr] = 1; // alapesetben tudjuk, tehat eltaroljuk
                fibStateRef.currentRes += 1;
                return Trampoline<FibonacciState>(fibStateRef, true);
            }

            if (fibStateRef.cache.find(cntr - 1) != fibStateRef.cache.end()) {
                if (fibStateRef.cache.find(cntr - 2) != fibStateRef.cache.end()) {
                    // ha n-1 re és n-2 re is tudjuk, akkor tudjuk n-re is!
                    fibStateRef.cache[cntr] = fibStateRef.cache[cntr - 1] + fibStateRef.cache[cntr - 2];
                }
                fibStateRef.currentRes += fibStateRef.cache[cntr];
            }
            else {
                // ellenkezo esetben a deque ELEJERE szurjuk be a kisebb ertekeket, hogy elobb vegezzunk veluk
                fibStateRef.currentCounters.push_front(cntr - 2);
                fibStateRef.currentCounters.push_front(cntr - 1);
            }

            return Trampoline<FibonacciState>(fibStateRef, true);
        }
    }
```

Monadok

A használat így néz ki:

```
int main()
{
    std::cout << "fibonacci(3) and fibonacci(4) are " << fibonacci(3) << ", " << fibonacci(4) << std::endl;
    std::cout << "fibonacci(10) = " << fibonacci(10) << std::endl;

    FibonacciState state(10, 0);
    Trampoline<FibonacciState> fibonacci10(state); // state-et REFERENCIAKENT adjuk at!
    fibonacci10.flatmap(fibonacciWorkhorse);
    std::cout << "fibonacci(10) = " << state.currentRes << std::endl;

    state.currentCounters.push_front(100000);
    state.currentRes = 0;
    Trampoline<FibonacciState> fibonacci100000(state);
    fibonacci100000.flatmap(fibonacciWorkhorse);
    std::cout << "fibonacci(100000) = " << state.currentRes << std::endl;
}
```

Monadok

Ha mindezt kipróbáljuk 500-as bemenettel, látható, hogy a naív algoritmus beláthatatlan ideig fut, a monadikus megoldás viszont pillanatok alatt visszatér.