S2-02 Multiparadigma programozás és Haladó Java 1

Tartalom

1. Memóriakezelés: referencia- és érték-szemantika

Ebben a fejezetben "referencia" alatt C++ referenciákat és pointereket értünk. Azaz bármit ami memóriacímet tárol (*Java-ban is referenciának hívják a pointereket*)

Változók viselkedésének két koncepciója

1. Hatáskör (scope)
   - Megmondja, hogy a program szövegében hol van egy bizonyos azonosító (*identifier*, lehet változó vagy függvény is)
   - Meghatározza a láthatóságot is: hol használható az azonosító
2. Élettartam (life)
   - Meghatározza meddig biztonságos egy memóriahelyen tárolni az értékeinket futásidő alatt
   - Az élettartam lejárta után azt a memóriahelyet már nem biztonságos olvasni és írni

Normális változónak: hatásköre és élettartama van

Referenciának: csak hatásköre van, élettartamot nem követ

Érték-szemantika (*Value semantics*)

- Értékadáskor magának az értéknek a másolása a változóba
- C++-ban érték-szemantika az alapértelmezett

Előnyök:

- Nincsenek memóriakezelési problémák
  - nincsenek "csellengő" (dangling) pointerek nemlétező memóriahelyre
  - nincsenek drága heap allokációk

  - nincsenek memóriaszivárgások

Referencia-szemantika (*Reference semantics*)

- Értékadáskor a memória címének másolása a változóba
  - vagy pointer másolása
- Referencia és pointer közötti különbség: pointer lehet `NULL`, referencia nem

Előnyök:

- Polimorfizmus megvalósítása
- Egyes esetekben jobb teljesítmény az érték-szemantikánál
- Nem mindent engedünk meg értékként másolni, például `std::cout`, `std::unique_ptr` és `std::shared_ptr`

Visszatérés referenciával

- C++-ban alapértelmezésben a függvény értékkel tér vissza, amit másol
- Visszatérhetsz referenciával is ha a megjelölöd a függvény típusában
- **LÁBON LŐHETED MAGAD**: ne térj vissza lokális változók címével, mivel azok élettartama a függvény scope-jához kötöttek
  - Emlékezz: referenciák nem követnek élettartamot

```
#include <iostream>

int& f()            // Függvény ami referenciát ad vissza
{
    int i = 300;    // Lokális változó a stack-en
    return i;       // Visszatérés lokális i változó címével
}   // HIBA: mire f függvény végetért, lokális i változó kiment a scope-ból
    //       és törlődött!

int main()
{
    int& x = f();

    // FUTÁSIDEJŰ HIBA: az alábbi sor "0"-t fog kiírni "300" helyett!
    std::cout << x << std::endl;

    return 0;
}
```

Copy elision

- Fordító képes kioptimalizálni az objektumok felesleges másolását

- C++98: Copy-konstruktorok elhagyása
- C++11: Move-konstruktorok elhagyása
- Másolásmentes értékszerinti átadás szemantika megvalósítása (*zero-copy pass-by-value*)
- Akkor alkalmazható, ha a változó egy temporary objektumból lett konstruálva
  - ide tartoznak a függvényparaméterek is

1. Inicializációkor

```
T x = T(T(T()));     // Ehelyett T x = T()

// Nem fogja a rakás copy-konstruktort meghívni,
// csak egy alapértelmezett konstruktort.
```

2. Függvényhívásban

```
T f() { return T(); }   // Visszatérés temporary-val

int main()
{
    T x = f();              // Ehelyett T x = T()
    T* p = new T(f());      // Ehelyett T* p = new T()
}
```

2. Referenciakezelési technikák, Objektumok másolása, move-szemantika

Referenciakezelési technikák

Memóriakezelés

Memória szegmens: Operációs rendszer ⇒ minden programnak egy területet tart fenn a memóriából.

Memóriacím: Minden bájt (memória hely) rendelkezik egy sorszámmal. Ezen keresztül elérhető a memória szegmensben. (Általában hexadecimális, pl.: `0x34c420`)

- Minden változó rendelkezik memóriacímmel
- C++-ban változókhoz hasonlóan kezelhetjük őket
- Típustól függően több bájton is tárolódhat egy változó értéke ⇒ mindig az első bájt címét kapjuk vissza

Referencia

Egy változó memóriacímét az `&` operátorral kérdezhetjük le, ez a referencia operátor. (`&<változónév>` a változó első bájtjának memóriabeli címe).

- Referenciákat eltárolhatjuk változókban
  - ezzel alias lesz egy változónévhez (mint Linuxban a szimbolikus linkek)

```
int i = 128;
int j = i;  // egyszerű változó
int& k = i; // referencia változó
```

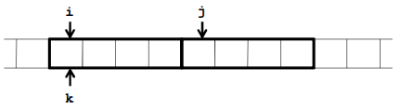

Figure 1: Referencia változók

Pointerek

Speciálisabb változótípus: memóriacímet tárol értékként.

- új adat, mely másik adat memóriacímét tárolja
- általánosabb célú, mint a referencia
- Létrehozáskor: megadjuk a mutatott érték típusát
  - mutató létrehozása: `<típus>* <mutatónév>`
  - mutató típusa: `<típus>*`
  - pl.: `int* ip; // egy int-re mutató pointer`
- pointer lehet `NULL`, referencia nem
- Ökölszabály:
  - ahol lehet használj referenciákat
  - ahol muszáj, ott pointereket (de akkor már smart pointert a mai világban)

Memóriaterületek

Használat szempontjából háromféle memóriaterületet különböztetünk meg:

- *globális terület*
  - konstansok, globális változók, statikus lokális változók
  - program indulásakor le van foglalva nekik terület (még ha nincsenek is inicializálva)
  - program futásának végéig jelen vannak
- *stack (verem)*
  - lokális változók
  - automatikusan jönnek létre definiáláskor, lokális scope végén automatikusan felszabadulnak
  - függvények memóriaterületeit stack frame-eknek hívjuk

1. függvényhíváskor hívó függvény (caller) stack frame-jének elmentése és hívott függvény (callee) stack frame-jének beállítása (function prologoue)
2. paraméterek mozgatása regiszterekbe
3. függvény végrehajtása
4. végén return value írása regiszterbe
5. hívott függvény stack frame pop-olása és visszatérés hívó függvénybe (function epilogue)

- *heap (kupac, free store)*
  - futásidőben dinamikusan lefoglalható (általában a legnagyobb része a programnak)
  - akkor foglaljunk a heap-en, amikor változó mérete csak futásidőben határozható meg, mérete stack overflow-t eredményezne vagy polimorfizmust szeretnénk megvalósítani
  - lassabb elérni, mint a stack-et

Memóriahely felszabadítás

A lefoglalt memóriát fel is kell szabadítani.

- automatikusan lefoglalt memória ⇒ program automatikusan végzi, nincs befolyásunk
- manuálisan létrehozott memória ⇒ nekünk kell törölni
- törlés:
  - `delete` operátorral
  - tömböket a `delete[]` operátorral (Ha tömbre is a `delete`-et használjuk, csak az első elem törlődik)
- Nem a mutatót, hanem a dinamikusan lefoglalt területet kell felszabadítani
- Több mutató hivatkozik ugyanarra a területre ⇒ elég egyszer törölni
- Nem szabad összemosni: C++-os `new`, `delete` ≠ Klasszikus C-s `malloc`, `calloc`, `realloc`, `free`
  - Klasszikus C-s memóriafoglalás: alacsonyszintű, bájtok közvetlen manipulálása memórián (még csak nem is operátorok, hanem függvények a Standard C könyvtárban)
  - C++-os memóriafoglalás: magasabb, objektumszintű, konstruktor és destruktor meghívása (rendes C++ operátorok, ám felül is lehet őket definiálni). Több lépésből állnak.
    - `new`: 1. Memória lefoglalása 2. Konstruktor meghívása
    - `delete`: 1. Destruktor meghívása 2. Memória felszabadítása
  - (*Megjegyzés: realloc alternatívája az std::vector használata*)

Konstans mutatók, referenciák

Módosíthatatlanná tehetjük a referenciákat a pointereket és az értékeket is:

(Trükk: Értelmezzük fordított sorrendben a deklarációkat)

```
double d1 = 10, d2 = 50;
double const_reference &d1r = d1;                          // konstans referencia
double const * pointer_to_const = &d1;                     // mutató konstansra
double * const const_pointer = &d1;                        // konstans mutató
// konstans mutató konstans értékre
double const * const const_pointer_to_const_value = &d1;

const_reference = 100;                    // HIBA, az érték nem módosítható
*pointer_to_const = 50;                   // HIBA, az érték nem módosítható
*const_pointer = 50;                      // az érték módosítható
*const_pointer_to_const_value = 50;       // HIBA
pointer_to_const = &d2;                   // átállíthatjuk más memóriacímre
const_pointer = &d2;                      // HIBA, a mutató nem állítható át
const_pointer_to_const_value = &d2;       // HIBA
```

Konstruktor, Destruktor

Típusok mezői is lehetnek mutatók, melyeknek dinamikusan allokálhatunk memóriaterületet. Ezt a *konstruktorban* végezzük.

A törlésről viszont gondoskodnunk kell. Ezt megtehetjük a *destruktorban*

- A destruktor automatikusan lefut, ha a változó törlésre kerül
  - lokális változó ⇒ blokk végén automatikusan törlődik
  - dinamikus létrehozás esetén a `delete` váltja ki a destruktor meghívását
- A destruktorban csak a dinamikusan lefoglalt mezőket kell törölni (ha ilyen nincs, akkor a destruktor nem szükséges)
- mindig publikus
- nincs típusa
- nincs paramétere
- nem túlterhelhető

```
class <típus> {
    public:
    <typusnév>() { ... } // konstruktor
    ~<típusnév>() { ... } // destruktor
    ...
};
```

Objektumok másolása

Kétféle másolási megközelítés ismert:

- *Sekély másolás (shallow copy)*: A típuspéldány a mezőivel együtt másolásra kerül egy új memóriaterületre. A dinamikusan lefoglalt mezőknek azonban

az értéke nem másolódik (A régi és új példány mutatói ugyanazon területre fognak mutatni.)

- *Mély másolás (deep copy)*: A típuspéldány minden mezőjével és azok által lefogalt memóriaterülettel együtt kerül másolásra. (A régi és az új példány mutatói nem ugyanazon területre fognak mutatni.)

Példányok másolását két művelet teszi lehetővé: *másoló konstruktor*, *értékadó operátor*

Copy-konstruktor

Egy létező példány alapján újat hoz létre. Paraméterként egy másik (ugyanolyan típusú) példány referenciáját kapja, ennek a mezőit másolja le. (Ha nincs dinamikus tartalom, akkor az alapértelmezett megfelelő.)

- törzsben tud hivatkozni a másolandó példány mezőire
- a következő esetekben fut le:
  - közvetlen hívás: `MyType b(a);`
  - kezdeti értékadás: `MyType b = a;`
  - érték szerinti paraméterátadás

```
class MyType {
    private:
        int* _value;
    public:
        MyType(const MyType& other) { // másoló konstr.
            _value = new int;

            // a dinamikus taralom létrehozása
            *_value = *other._value; // érték másolása
        }
};
```

Értékadó operátor

A kezdeti értékadást kivéve, amikor a változónak értéket adunk, az értékadó operátor lép érvénybe.

Megkapja a másolandó példány (konstans) referenciáját, és biztosítja taralmának átmásolását.

- az eddig meglévő, dinamikusan létrehozott értékeket törölni kell
- ellenőrizni kell, hogy a paraméterben kapott változó nem saját maga-e
- a `*this` (aktuális példány) referenciával kell visszatérni (a többszörös értékadás használatához)

```
class MyType {
    public:
```

```
    ...
        MyType& operator=(const MyType& other){
            if (this == &other)
                // ha ugyanazt a példányt kaptuk
                return *this; // nem csinálunk semmit

            *_value = *(other._value);
            // különben a megfelelő módon másolunk
            return *this; // visszaadjuk a referenciát
        }
};
```

Paraméterátadás

Ahogy a változókat, úgy a paramétereket is háromféleképpen tudjuk átadni:

- **érték szerint**
- **referenciaként**
- **pointerként**

Az érték szerinti átadás sokszor költséges lehet, mert ekkor a paraméterek másolódnak. A pointer és a referencia közötti döntést pedig az határozza meg, hogy míg a pointerek felvehetik a `NULL` értéket, addig a referenciák nem.

Tehát a következők szerint érdemes a paraméterátadást használni:

- *érték szerinti*: `f(int x)`
  - a függvény **nem módosíthatja a paramétert**
  - használd ezt, **ha könnyű másolni**
  - Ökölszabály: ha a paraméter mérete legfeljebb 2- vagy 3 szó (word), azaz 32-bit, akkor érdemes érték szerint átadni
    - egyszerű primitív típusok esetén ajánlott, mint pl.: `int`, `double`, `char`, `bool`, stb.
    - Komplex típusok, saját osztályok, `std::string` és a különböző STL konténerek nem ajánlottak (továbbiakban ezeket a példákban `T`-vel jelöljük)
- *pointer szerinti*: `f(T* x)`
  - a függvény **módosíthatja a paramétert**,
  - használd ezt, **ha költséges a másolás**,
  - továbbá ha a **NULL lehet valid** érték
- *konstans pointer szerinti*: `f(const * T x)`
  - a függvény **nem módosíthatja a pointer által mutatott értéket**,
  - használd ezt, **ha költséges a másolás**,
  - továbbá ha a **NULL lehet valid** érték
- *referencia szerinti*: `f(T& x)`
  - a függvény **módosíthatja a paramétert**
  - használd ezt, **ha költséges a másolás**,

    - továbbá ha a **NULL NEM lehet valid** érték
- *konstans referencia szerinti*: `f(const int& x)`
    - a függvény **nem módosíthatja a paramétert**,
    - használd ezt, **ha költséges a másolás**
    - továbbá ha a **NULL NEM lehet valid** érték
    - próbáljuk mindig ezt a változatot használni saját osztályokhoz, `std::string`-hez és STL adatszerkezetekhez, ha paramétert nem akarjuk módosítani
        * copy konstruktorok, copy assignmentek paraméterei

Move-szemantika

A C++-ban értékszemantika van. Ez egy tiszta memóriaterület szeparációt tud eredményezni, de sokszor teljesítményromlást okozhat nagy objektumok másolása esetén.

Tekintsük a következő `Array` implementációt:

```
class Array{
    public:
        Array (const Array&);
        Array& operator=(const Array&);
        ~Array ();
    private:
        double *val;
};
Array operator+(const Array& lhs, const Array& rhs){
    Array res = left;
    res += right;
    return res;
}
```

Az `Array` egy osztály, melynek `+` operátora összekonkatenálja a két paramétert és visszaad egy új listát.

A következő függvény meghívásánál azonban több köztes `Array` példány keletkezik és szűnik meg:

```
void f()
{
 Array b, c, d;
 ...
 Array a = b + c + d;
}
```

A move-szemantika az ehhez hasonló problémákra ad megoldást.

- másolás helyett “ellopja” az erőforrást

- törölhető állapotban hagyja a másik objektumot
- teljesítményoptimalizáció, objektum mozgatása másolás nélkül
- objektum birtokosának megváltoztatása (*ownership transfer*)
  - például `std::unique_ptr`-t nem szabad másolni, különben ki lesz az erőforrás birtokosa akinek a scope-jából kimegy?

Ehhez kell:

- overloadolni lehessen a *Copy-konstruktort* és az *értékadó operátort*, illetve egyéb függvényeket
  - meg kell tartani a backward compatibility-t
  - meg kell különböztetni a bal- és jobbértékeket

RValue, LValue

Korábbi nyelvekben értékadás: `<variable> = <expression>` (pl.: `x = 5`)

C/C++-ban értékadás: `<expression> = <expression>` (pl.: `*++ptr = *++qtr`)

- de nem minden esetben működik, pl.: `a+5 = x` helytelen.

Lvalue:

- kifejezés, mely értékadás után is létezik
- azonosítóval egy memóriaterületre hivatkozik
- minden változó és konstans változó Lvalue
- lehetővé teszi, hogy a `&` operátorral (*Lvalue referencia operátor*) megszerezzük annak memóriacímét

Rvalue:

- átmeneti (temporary) kifejezés, ami értékadás után már nem létezik
- van értéke, de nem lehet értéket rendelni hozzá
- literálok (`5`) és aritmetikai kifejezések Rvalue-k

Rvalue referencia operátor (&&)

A `&&` operátorral lehet kasztolni Lvalue-t Rvalue-vá. (balértékből jobbértéket)

- kikényszeríthetjük a move-szemantika használatát

Példa:

```
struct S
{
 S() { a = ++cnt; std::cout << "S()" << std::endl; }
 S(const S& rhs) { a = rhs.a; std::cout << "copyCtr" << std::endl; }
 S(S&& rhs) { a = rhs.a; std::cout << "moveCtr"<< std::endl; }
 S& operator=(const S& rhs) { a = rhs.a; std::cout << "copy=" << std::endl; return *this; }
 S& operator=(S&& rhs) { a = rhs.a; std::cout << "move=" << std::endl; return *this; }
```

```
 int a ;
 static int cnt;
};
int S::cnt = 0;

int main()
{
 S a, b;
 swap( a, b);
}
```

Move-operátor használata nélkül:

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

Eredmény:

```
S()       // S a
S()       // S b
copyStr   // T tmp(a)
copy=     // a = b
copy=     // b = tmp
```

Move-operátor használatával:

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

Eredmény:

```
S()       // S a
S()       // S b
moveCtr   // T tmp(std::move(a))
move=     // a = std::move(b)
move=     // b = std::move(tmp)
```

Perfect forwarding

- Függvény kapott paraméterének továbbítása egy másik függvény paraméterének
  - úgy, hogy az érték megtartja kategóriáját (Lvalue vagy Rvalue)
- Megvalósítás `std::forward`-al

```
template<class T>
void wrapper(T&& arg)
{
    // arg mindig Lvalue lesz
    foo(std::forward<T>(arg)); // T-től függően arg továbbítása Lvalue- vagy RValue-ként
}
```

Például saját `make_unique` írása (`unique_ptr` nem másolható):

```
template<class T, class U>
std::unique_ptr<T> my_make_unique(U&& u)
{
    return std::unique_ptr<T>(new T(std::forward<U>(u)));
}
```

3. Erőforrásbiztos programozás, RAII, destruktor és szemétgyűjtés

Resource Allocation is Initialization-elv (RAII)

- Erőforrások: fájlok, mutexek, socket-ek
- Erőforrás becsomagolása egy objektumba, ahol
  - Konstruktor: megszerzi, lefoglalja az erőforrást, beállítja az osztály invariánsait
  - Destruktor: elengedi, felszabadítja az erőforrást (nem dob kivételt!)
- Az erőforrás élettartama az azt "becsomagoló" objektum élettartamához, scope-jához kötött
  - Ha az objektum kimegy a scope-ból $\Longrightarrow$ destruktor automatikusan felszabadítja az erőforrást
  - Szokás ezt az objektumot az erőforrás "birtokosának" is hívni
  - Hasznos kerülni a globális változókat birtokosként, minél inkább lokálisabbra korlátozzuk a scope-ot, annál jobb
- Kivételbiztos (*exception safe*) programozás támogatása

Példák:

- `std::ifstream`, `std::ofstream`
  - Konstruktor: fájl megnyitása
  - Destruktor: fájl lezárása
- `std::string`

  - Konstruktor: `char` tömb dinamikus allokációja
  - Destruktor: `char` tömb felszabadítását
- STL konténerek
- `std::lock_guard`
  - Konstruktor: kölcsönös kizárás megvalósítása többszálú környezetben, mutex becsomagolása és lock-olása
  - Destruktor: mutex elengedése
- RAII osztály írása hálózati socket-ekhez amik lezárják magukat

Smart pointerek általában

- RAII filzófiát terjesszük ki a memóriára
  - nem memóriában kell gondolkodni
  - memóriát úgy kezeljük, mint általános erőforrást
- `new` elrejtése konstruktorban, `delete` hívása destruktorban amikor smart pointer elhagyja a scope-ot
- Destruktorban automatikusan felszabadítja a becsomagolt objektumot a heap-ről
- Kivételkezelésnél hasznos
  - pre-C++11: mi van ha `new` és `delete` hívása között kivétel keletkezik és `delete` soha nem hívódik meg?
- Felüldefiniált operátorokkal (`*`, `->`) nyers pointer-ként használható
- `get()`: nyers pointer visszaadása

Auto pointer (std::auto_ptr)

- Legegyszerűbb smart pointer, ownership-modell alapján kezeli a memóriát
  - `auto_ptr`-t birtokló objektum scope-jához kötött
- Copy konstruktor és copy assignment hívásakor nincs másolás $\Longrightarrow$ ownership átadása
- Még C++98-ból maradt meg
- Elavult, nem ajánlott
  - STL konténerekkel és algoritmusokkal nem működik jól
  - Tömbökkel nem működik együtt
  - Ownership átadása másoló műveletek felüldefiniálásával nem az igazi
  - Nem lehet tetszőleges deleter-t adni neki

Unique pointer (std:unique_ptr)

- `auto_ptr` "javított változata" C++11-től, ajánlott
- STL-el jól működik
- *move-only* típus, nem másolható
  - Copy konstruktora és copy assignment operátora le van tiltva

  - Helyette van move konstruktora és move assignment operátora $\Longrightarrow$ igazi ownership átmozgatás
- Template paraméterként tetszőleges deleter megadása
- Nincs overhead-je, ugyanannyit foglal mint egy nyers pointer
  - kivéve ha deleter-t adunk
- Érdemes `std::make_unique`-al foglalni memóriát
  - C++14-től
  - "No news means good news": explicit `new` operátor hibát dobhat

Shared pointer (std::shared_ptr)

- Megosztott ownership több birtokos objektum között
- Referenciaszámlálót használó pointer
  - scope elhagyásakor számláló csökkentése eggyel
  - amikor az utolsó `shared_ptr` kimegy a scope-ból és a referenciaszámláló nulla lesz: felszabadítás
- `std::make_shared` (C++11-től): mint `std::make_unique`
- Létrehozáskor egy külön memóriaterületen sharing group, control block létrehozása
  - ez a blokk tartalmazza a referenciaszámlálót és a menedzselt objektumot
  - szálbiztos hozzáférés, referenciaszámláló növelésekor lock-olás
  - ezért lassabb `unique_ptr`-nél, ha lehet inkább `unique_ptr`-t érdemes használni (tervezd meg jobban a szoftvered)
  - (*incidens: Java programozók áttértek C++11-re, telerakták std::shared_ptr-el az egész programot és lassabban működött mint Java garbage collector-ja*)
- `shared_ptr`-el lábon lőheted magad: ciklikus referenciák $\Longrightarrow$ memóriaszivárgás. Megoldások:
  - `unique_ptr` használata és ownership mozgatása új birtokosnak
  - `weak_ptr` használata

Weak pointer (std::weak_ptr)

- Referencia tárolása objektumra, amit `shared_ptr` már kezel
  - Egyfajta passzív megfigyelő, "observer"
  - Nem birtokol semmit
- Nincs memóriakezelő művelete
- "Ellenszer" ciklikus `shared_ptr` referenciákra
- Ha kell, `shared_ptr`-é konvertálható
- Ha `shared_ptr` törlődik: üres helyre mutat!

Szemétgyűjtés (*Garbage Collection, GC*) Java-ban

(*Nem találtam Multiparadigma programozás tananyagában C++ garbage collector-okról szóló részt. Ha mégis kell beszélni valamit róla, akkor a Boehm garbage collector-t érdemes megemlíteni.*)

- Java-ban csak allokálni lehet, deallokálás nem megengedett
- Referenciatípusok csak heap-en tárolódnak
- Referenciafajták Java-ban:
  - Strong reference: alapértelmezett referencia, GC nem törölheti
  - Soft reference: ha egy objektumra minden referencia Soft, akkor nem garantált, hogy GC megtartja őket
  - Weak reference: mint Soft, csak GC előbb szabadítja fel
- Szemétgyűjtés: a futtatórendszer időnként detektálja és deallokálja azokat az objektumokat, amik már nem elérhetők, nincs rájuk referencia

Referenciaszámláló szemétgyűjtés (reference counting)

- Minden egyes objektumra számoljuk, hogy hányan hivatkoznak rá
- Gyors, csak a számlálót kell növelni és csökkenteni
  - `new`, allokáció: növelés eggyel
  - referencia megszerzése, hivatkozás az objektumra máshonnan: növelés eggyel
  - referencia elhagyása: csökkentés eggyel
- Ciklikus referenciák esetén viszont nem tud deallokálni, memóriaszivárgás történik (vesd össze: `std::shared_ptr`)

Mark and Sweep szemétgyűjtés

- Megjelöli (mark) az elérhető objektumokat, a jelöletlen objektumokat pedig törli (sweep)
- Gyökérhalmaz (root set): referenciák amik használatban vannak
- Mark and sweep két fázisa:
  1. Mark: gyökérhalmazból indulva a GC bejárja a referenciákat tartalmazó gráfot és megjelöli a használt objektumokat
  2. Sweep: végighalad a jelöletlen objektumokon és törli őket
- Naiv implementáció szünetelteti a programot szemétgyűjtés közben ("stop the world")

Generációs szemétgyűjtés (Generational)

- Heap több részre, "generációra" van osztva
  1. Fiatal generáció (Young generation)
  2. Idős generáció (Old generation)
  3. Végleges generáció (Permanent genertion)
- Minden új objektum a fiatal generációba kerül

- Minor garbage collection: amikor a fiatal generáció megtelik (gyorsabb)
- A fiatal generáció túlélői (survivors) "idősödnek", egy bizonyos "kor" elérése után pedig bekerülnek az idős generációba
- Major garbage collection: amikor az idős generáció megtelik (lassabb)
- Végleges generáció: JVM metaadatai tárolódnak amik az osztályokat leírják

4. Kivételkezelés, kivételbiztos programozás

Hibakezelés

Hiba: program futása alatt bekövetkezett nemkívánt állapot. Két fajtáját különböztetjük meg:

- Logikai hiba (logic error): a program nem az elvárásnak megfelelő eredményt nyújtja, elő- vagy utófeltétel megsértése
- Futásidejű hiba (runtime error): rendszer állapotától függő hibás végrehajtás vagy megszakítás, mint például memória elfogyása, hibás IO, hibás hálózati kapcsolódás

Hibakezelés alatt a runtime error-okkal foglalkozunk.

Hibakezelés klasszikus C-ben

- függvény visszatérési értékei hibakódok / NULL referencia
- externális `errno` kód
- visszatérési érték vizsgálat, pl.: `fseek`, `int`-et ad, `bool`-ként kezeljük
- hiba flag-ek, pl.: hibaflag beállítódik ⇒ többi IO művelet nem csinál semmit
- `assert()`: nem teljesül a feltétel ⇒ hibaüzenet, mely a felhasználó számára nemigazán érthető (pl. egy repülőpilótának nem bizalomgerjesztő egy ilyen üzenet: `Assertion failed: Inv() line 64`)
- hibakód változójának paraméterként való átadása referenciaként (klasszikus C-ben pointer)
- struct-al való visszatérés, mely az eredmény mellett tartalmazza a hibakódot is (mintha egy pair-ünk lenne)

Kivételkezelés alapköve C-ben: jump műveletek

- Call-stack állapotát lehet a módszerrel elmenteni/visszaállítani
  - függvényhíváskor ⇒ függvény stackframe a stack-re kerül
  - a jump műveletekkel vissza lehet állni a stack-en korábbi állapotra (a bázispointer helyének segítségével)

1. `jmp_buf x;` - reprezentálja az elmentett stack állapotot
2. `setjmp(x)` - elmenti a stack állapotát, illetve ide ugrik vissza a vezérlés `longjmp` hívása után

3. `longjmp(x, 5)` - kiváltja a stack visszaállítását az `x` által reprezentált állapotba. Mellékel egy hibakódot is, melyet a `setjmp` visszaad.

Használat:

```
#include <setjmp.h>
#include <stdio.h>

jmp_buf x;

void f()
{
    longjmp(x,5);
}

int main()
{
    int i = 0;

    if ( (i = setjmp(x)) == 0 )
    {
        f();
    }
    else
    {
        switch( i )
        {
        case  1:
        case  2:
        default: fprintf( stdout, "error code = %d\n", i); break;
        }
    }
    return 0;
}
```

Érezhetőek a következő megfeleltetések:

- `setjmp` - `try`
- `longjmp` - `throw`
- `else` - `catch`

Static Assert (C++11)

Fordítási időben kiszámolható boolean kifejezések (`bool_constexpr`) igazságértékét vizsgálja

- ugyanúgy, mint a sima `assert()`, hamis esetben hibát dob, de jelen esetben forítási hibát.

Kivételkezelés

Kivételkezelés célja

- elválasztani a hiba fellépésének és kezelésének helyét (én detektálhatom a hibát, de másvalaki más modulban tudja mit csináljon vele)
- az adatot típushelyesen tudjuk a hiba fellépési helyétől a handler-hez szállítani
- ne járjon semmilyen extra (kód/idő/hely) hátránnyal, ha nem használjuk
- minden kivételt a megfelelő handler kapja el
- többszálú környezetben is megfelelő
- ha nincs hiba ⇒ ne legyen overheadje (vagy legalábbis minimális legyen)
  - `if`-ek nem jók, mert a folyamatos kiértékelés órajelet emészt
  - kivételek megcsinálhatók úgy, hogy majdnem költségmentesek legyenek

try-catch

```
try {
    f();
    // ...
}
catch (T1 e1) { /* handler for T1 */ }
catch (T2 e2) { /* handler for T2 */ }
catch (T3 e3) { /* handler for T3 */ }
```

- `throw` - bármi dobható, de az értelmes: `std:exception` leszármazottjai
  - Futásidejű hibákhoz: `std::runtime_error`
  - Logikai hibákhoz: `std::logic_error`
- `catch(T e)` - `T` típusú érték handlere
  1. Elkapja a kivételt, ha az `T` típusú
  2. vagy annak leszármazottja
  3. illetve pointer vagy referencia és a hivatkozott értékre fennáll 1) vagy 2)
- Nem szabad `new`-val exception-t létrehozni ⇒ memóriaszivárgás

Hierarchia

- Az öröklődést használjuk kivételek csoportosítására
- Az általánosabb handler elkapja a speciálisabb exceptiont
- A `catch` ágak a megadott sorrendben értékelődnek ki
- ezért figyelni kell a handlerek általánosságát és sorrendjét
  - A speciálisabb kerüljön felülre és az általánosabb alulra

Továbbra sem ajánlott `new`-val kivételt létrehozni. Ha a dianamikus típussal akarunk játszani, inkább használjuk a következő megoldást:

```
struct ExceptionBase{
    virtual void raise() { throw *this; }
    virtual ~ExceptionBase() {}
};

struct ExceptionDerived : ExceptionBase{
    virtual void raise() { throw *this; }
};

void foo(ExceptionBase& e){
    e.raise(); // Uses dynamic type of e while raising an exception.
}

int main (void){
    ExceptionDerived e;
    try {
        foo(e);
    }catch (ExceptionDerived& e) {
        ...
    }catch (...) {
        ...
    }
}
```

Kivételkezelés és osztályok

Kérdéses esetek: konstruktor, destruktor

- Konstruktor
  - Ha a konstruktor dob, terület le lett foglalva, de a pointer nem lett beállítva ⇒ nincs gond, a rendszer deallokálja
  - De: a konstruktoron belül lefogalt területet a destruktor tudja felszabadítani. A destruktort viszont nem lehet meghívni mert az objektum létre se jött rendesen ⇒ probléma

```
class X
{
    public:
        X(int i) { p = new char[i]; init(); }
        ~X() { delete [] p; }       // must not throw exception
    private:
        void init() { ... throw ... }   // BAD: destructor won't run !
        char *p;                        // constructor was not completed
    };
```

Ha tagváltozó inicializálása dob hibát, akkor dob a konstruktor is

```
class X
{
    public:
        X() { throw 1; }
};
class Y
{
    public:
        Y()
        try
            : x()
        { }
        catch( ... ) { /* throw; */ }
    private:
        X x;
};
int main(){
    try {
        Y y;
        return 0;
    }
    catch (int i)
    {
        std::cerr << "exception: " << i << std::endl;
    }
}
```

- Destruktor
  - destruktorokat kétféle okból hívunk
    - nomális esetben
    - kivételkezeléskor ⇒ ha a destruktor is kivételt dob az nem definiált viselkedéshez vezet (legtöbbször a `terminate()` meghívásához)
  - ⇒ Ökölszabály: **destruktor nem dobhat kivételt**

Noexcept (C++11)

- Kifejezhető vele, hogy egy kifejezés, függvény biztosan nem dob-e exceptiont
- Fordítási időben értékelődik ki

Kétféle formában létezik:

- operátor forma: `bool noexcept(expr);`
  - Nem értékeli ki a kifejezést (hasonló a `sizeof`-hoz)
  - `false`, ha
    - a kifejesés dob

        * a kifejesésben van dynamic_cast
        * a kifejesésnek van `type_id`-ja
        * van a kifejesésben függvény, ami nem `noexcept(true)` és nem `constexpr`
    - `true` különben
- specifier forma: `void f() noexcept(expr) { }`
    - a régi `throw()` helyett van
    - pl.: Ha `g()` nem dob, akkor `f()` sem:

```
template <typename T>
void f() noexcept ( noexcept( T::g() ) )
{
  g();
}
```

Magyarázat: `noexcept( T::g() )` - operátor formás noexcept, megmondja, hogy `g()` dob-e exception-t

- Ha igen, akkor `void f() noexcept(false)` lesz fordítás után
- Ha nem dob, akkor `void f() noexcept(true)` fordítás után
- tehát ha `g()` nem dob, akkor `f()` sem

5. A konkurens programozás alapelemei Javában és C++-ban

Problémák a C++98 memóriamodellel

- Egyszálas vezérlésre tervezték
- Fordító kioptimalizálhat változókat, még `volatile` esetén is (az a kulcsszó is egyszálasra tervezett)
- Légből kapott értékeket kaphatunk

C++11 memóriamodell

- Új memóriamodell, standard könyvtár-beli támogatás szálkezelésre, szinkronizációra és atomikus műveletekre
- Rendszer garantálja, hogy a párhuzamos végrehajtás szekvenciálisan konzisztens lesz

Hogyan írjunk szálbiztos Singleton-t C++11-től?

C++11 memóriamodellje már garantálja, hogy lokális statikus változók szálbiztosan jönnek létre

```cpp
class Singleton   // Meyers Singleton, nevét Scott Meyers-ről kapta
{
public:
    Singleton(const Singleton&) = delete;
    Singleton(Singleton&&) = delete;
    Singleton& operator=(const Singleton&) = delete;
    Singleton& operator=(Singleton&&) = delete;
    static Singleton& getInstance() const
    {
        return _instance;
    }
private:
    Singleton() {}
    static Singleton _instance;
};
```

`std::thread`

- Konstruktorába függvénynevet, függvénypointert, funktort vagy lambdát adhatunk
  - továbbá függvények paramétereit
- Konstruktorhívás után új szálat indít, paraméterként kapott függvényt az új szálban hajtja végre
- `thread.join()`
  - Hívó szálat blokkolja, amíg a létrehozott szál nem végez
  - Ha függvény végzett és a szál destruktora meghívódott `join()` nélkül: `std::system_error` kivételt dob
  - Írhatunk akár egy saját RAII-elvű `scoped_thread` osztályt, ami becsomagolja `std::thread-et` és destruktorában meghívja `join()`-t
- `thread.detach()`
  - Megengedi hogy "leváljon" a száltól aki létrehozta
  - Ne várja meg a főszál, míg a detach-elt szál befejeződjön, függetlenül hajtódik végre
  - detach-elt szál futása végén felszabadítja erőforrásait és nem dob `std::systeM-error`-t

`std::mutex`

- Szinkronizációt segítő, kölcsönös kizárást megvalósító objektum amit a szál birtokol
- Műveletei:
  - `lock()`: lezárás, más szálat nem enged a kölcsönösen kizárt régióba. Szál blokkol amíg nem sikerül megkapnia
  - `unlock()`: elengedés, más szál lefoglalhatja a mutexet

  - `try_lock()`: mint `lock()`, de sikertelen lock-olás esetén csak `false`-al tér vissza
- Változatai:
  - `std::recursive_mutex`: a mutex birtokosa nem fogja saját magát blokkolni ha újra lefoglalja a mutexet
  - `std::timed_mutex`: megadható hogy amikor lock-olni akar, akkor mennyi ideig próbálkozzon

`std::lock_guard`

- RAII-csomagoló mutex számára
  - konstruktorban lezárja a mutexet
  - scope-ot elhagyva destruktorban elengedi a mutexet
- Nem másolható
  - `std::unique_lock`: csak move-olható

`std::atomic<T>`

- Csomagoló, a becsomagolt változóban atomikus írási és olvasási műveletek
- Nem kell állandóban `mutex.lock()`/`mutex.unlock()`-okat írni eléréséhez, csökkenti a boilerplate kódot
- Műveletei:
  - `store()` (atomikus írás)
  - `load()` (atomikus olvasás)

Párhuzamos programozás C++-ban (`std::async`, `std::future`, `std::promise`)

- `std::async`
  - Elndít egy aszinkron számítást új szálban
  - Eredményét `future`-be írja
  - Hiba esetén kivételobjektumot ír a `future`-be
- `std::future`
  - Csak olvasható
  - `get()`-el kapjuk meg az eredményt és addig blokkolja a vezérlést, amíg meg nem kapjuk

```
std::future<int> myfuture = std::async(std::launch::async, myfunction);

// ...

int x = myfuture.get()  // Blokkolni fog amíg std::async-ból
                        // indított szál nem végez és nincs eredmény
```

- `std::promise`
  - Egy külön szál írhatja
  - `future` készíthető belőle `promise.get_future()`-el
  - `promise` és `future` egy csatornát alkot, illetve termelő-fogyasztó munkamenet szimulálható
    * `promise`: író (termelő)
    * `future`: olvasó (fogyasztó)

Konkurens és párhuzamos programozás Java-ban

Szálak:

- Támogató osztály: `java.lang.Thread`
- Mindkét esetben a `run()` metódust kell felüldefiniálni
- Szál létrehozása:
  1. Saját szál származtatása a Thread osztályból: `new HelloThread().start()`
  2. `Runnable` interfészt megvalósító osztály létrehozása és `Thread` konstruktor paramétereként adása: `new Thread(new HelloRunnable()).start()`
- Szálak lehetséges életciklusa:
  1. Létrejött (created)
  2. Futtatható (runnable)
  3. Futó (running)
  4. Blokkolt (blocked)
  5. Végetért (terminated)

Alapvető szinkronizáció:

- `synchronized` metódus:
  - Szál annak az objektum lock-ján zárol, akié maga a `synchronized` metódus.
- `synchronized` blokk:
  - Természetesen nem csak metódusokra implikálható, szinkronizációs blokkot is hozhatunk létre, melynek belsejében adhatjuk meg azokat az utasításokat melyekre kizárólagos hozzáférést szeretnénk biztosítani.
- `volatile` változó:
  - Explicit lock nélküli szinkronizációt biztosít, rákényszerítve a fordítót, hogy mindig olvassa ki a volatile változó értékét s így nem fordulhat elő az, hogy elavult (cache-elt) értéket kapjunk.
- Konkurens használatra tervezett adatszerkezetek: `Vector`, `CopyOnWriteArrayList`, stb.
- Szinkronizációs osztályok: `Latch`, `Semaphore`, `Barrier`

Párhuzamos programozás

- `Future`: interfész, aszinkron számítás eredményét reprezentálja (mint C++11-ben az `std::future`)
  - Eredményt `get()`-el kaphatjuk meg, mely addig blokkolódik, míg meg nem kapja az eredményt.
  - Alapimplementációja `FutureTask`
- `Executor`
  - `Executor` egy szálat több `Runnable` objektum végrehajtására tud felhasználni. Elosztja a beérkező feladatokat egy pool-ban lévő szálak között.
  - `ExecutorService` egy kiterjesztése az `Executor`-nak. A taszkok beküldésekor (`submit()`) nem csak `Runnable`-t, hanem `Callable`-t is elfogad, ez `Future`-t tud visszaadni.

6. További források

- http://aszt.inf.elte.hu/~gsd/multiparadigm/
- https://isocpp.org/faq
- http://thbecker.net/articles/rvalue_references/section_01.html
- https://docs.oracle.com/javase/tutorial/essential/concurrency/
- http://www.cplusplus.com/articles/z6vU7k9E/
- http://people.inf.elte.hu/groberto/elte_amp/eloadas_anyagok/elte_amp_ea09_eml.pdf
- http://people.inf.elte.hu/groberto/elte_amp/eloadas_anyagok/elte_amp_ea10_eml.pdf
- https://isocpp.org/wiki/faq/value-vs-ref-semantics
- http://en.cppreference.com/w/cpp/language/copy_elision
- http://aszt.inf.elte.hu/~gsd/multiparadigm/3_ptr_ref/ptrref4.cpp.html
- https://docs.microsoft.com/en-us/cpp/cpp/lvalues-and-rvalues-visual-cpp
- https://github.com/AnthonyCalandra/modern-cpp-features
- http://en.cppreference.com/w/cpp/language/raii
- https://isocpp.org/wiki/faq/cpp11-library#unique-ptr
- http://kitlei.web.elte.hu/segedanyagok/foliak/java/en-java-bsc/02object-orientation.pdf
- http://www.oracle.com/webfolder/technetwork/tutorials/obe/java/gc01/index.html