C++

5. HÉT – SMART POINTERS

A JEGYZETET KÉSZÍTETTE:

CSAPÓ ÁDÁM BALÁZS ÉS ŐSZ OLIVÉR

A C++ mögött máig aktív szabványosító testület áll – minden 3-4 évben új szabvány jelenik meg.

Szerencsére a 'gyártók' (akik a fordítókat készítik) a szabványt komolyan is veszik.

AZ ELSŐ NÉHÁNY HÉTBEN A C++11 ÉS AZÓTA MEGJELENT ÚJÍTÁSOKRÓL ESIK SZÓ

Dinamikus memóriakezelés lehetséges hibái

Ha dinamikusan foglalunk memóriát, akkor nekünk kell gondoskodni adnak felszabadításáról

Memory leak (szivárgás): nem kerül felszabadításra a már nem használt memória

Double free: többszörösen felszabadításra kerül egy blokk

Use after free: már felszabadított memóriához próbálunk hozzáférni

Dangling pointer/reference: már felszabadított/érvénytelen memóriaterületre hivatkozó pointer vagy referencia

- Ez akkor is előfordulhat, ha nem foglalunk dinamikus memóriát, csak lokális változókra hivatkozunk, amik megszűnnek

Mikor van szükség dinamikus memóriakezelésre és pointerekre?

Ahol nem muszáj, ne használjunk dinamikus memóriát, mert a nagyobb hibalehetőség mellett a program teljesítményét is rontja a sok foglalás

Bizonyos esetekben viszont nem lehet elkerülni:

- Futás közben derül ki, mennyi memóriára van szükség (pl. adatok beolvasása miatt)
- Sok memóriára van szükség, ami nem fér bele a stack memóriába
- Objektumok, függvények egymásnak adják át a létrehozott adatokat (és nem akarjuk azokat másolni)
- Polimorf objektumok kezelése

Smart pointerek

A **smart pointerek** megoldást kínálnak az említett problémákra. Olyan osztályok, melyek úgy viselkednek, mint a sima (raw) pointerek, de segítenek a mutatott memóriaterület kezelésében.

Házilag is könnyen készíthetünk egy olyan wrapper osztályt, ami a destruktorában felszabadítja a benne tárolt pointert:

```
class SmartPtr {
  int *ptr; // Maga a pointer
public:
  explicit SmartPtr(int *p = NULL) { ptr = p; }
  ~SmartPtr() { delete(ptr); }
  int &operator *() { return *ptr; }
};
```

```
int main() {
  SmartPtr ptr(new int());
  *ptr = 20;
  std::cout << *ptr;
  // Nem kell delete ptr, mivel az osztalynak van destruktora...
  return 0;
}
```

A memóriaterület automatikusan felszabadul, amikor a SmartPtr megszűnik. Nem marad el a felszabadítás, akkor se ha exceptiont dob a kód, és nem kell kézzel meghívni a delete-et.

Házi smart pointer

Generikus változat T típusra, és -> operátorral a tagok eléréséhez:

```
template <class T>
class SmartPtr {
T *ptr; // Maga a pointer
public:
  explicit SmartPtr(T *p = NULL) { ptr = p; }
  ~SmartPtr() { delete(ptr); }
  T & operator * () { return *ptr; }
  T * operator -> () { return ptr; }
};
```

```
int main()
{
  SmartPtr<int> ptr(new int());
  *ptr = 20;
  std::cout << *ptr;
  return 0;
}
```

std:: smart pointerek

Természetesen a smart pointer osztályt nem kell nekünk definiálni, már eleve van többféle a sztenderd könyvtárban.

2011-ig volt az auto_ptr osztály, aztán ez deprecated lett és 2017-ben teljesen eltűnt a nyelvből.

Van viszont **unique_ptr**, **shared_ptr** és **weak_ptr** – mindegyik az std névtérben a <memory> headerben.

Nézzük elsőként a unique_ptr típust!

Ez a típus garantálja nemcsak a pointer becsomagolását / megfelelő felszabadítását, hanem azt is, hogy más változó nem hivatkozhat ugyanarra a mem.területre. Magyarán: egy unique_ptr objektumot nem lehet másolni (std::move-val viszont lehet mozgatni).

unique_ptr

```
std::unique_ptr<int> p1(new int(5));
//std::unique_ptr<int> p2 = p1;  // Compile error: attempting to reference deleted function
std::unique_ptr<int> p3 = std::move(p1);  // Atadja a mem.teruletet. A mem.terulet most mar p3-e, es p1 nullptr.

if (p1 == nullptr) { std::cout << "nullptr indeed" << std::endl; /* ezt kiirja */ }
// std::cout << *p1 << std::endl; // segmentation fault!
std::cout << *p3 << std::endl;
p3.reset();  // Torli a memoriat
if (!p3) { std::cout << "nullptr indeed" << std::endl; /* ezt kiirja */ }
//std::cout << *p3 << std::endl; // segmentation fault!
p1.reset();  // Semmit nem csinal
```

unique_ptr

A reset() metódusnak adhatunk új memóriacímet is – ilyenkor a továbbiakban már azt fogja menedzselni, a régi mem.területet felszabadítja:

```
std::unique_ptr<int> p1(new int(5));

//std::unique_ptr<int> p2 = p1;  // Compile error: attempting
to reference deleted function

std::unique_ptr<int> p3 = std::move(p1);  // Atadja a
mem.teruletet. A mem.terulet most mar p3-e, es p1 nullptr.



if (p1 == nullptr) { std::cout << "nullptr indeed" <<
std::endl; /* ezt kiirja */ }

// std::cout << *p1 << std::endl; // segmentation fault!

std::cout << *p3 << std::endl;
```

```
p3.reset(new int(4));  // Torli a
memoriat, de ezentul az ujonnan
lefoglaltat menedzseli

if (p3 == nullptr) { std::cout <<
"nullptr indeed" << std::endl; /* ezt NEM
irja ki */ }

std::cout << *p3 << std::endl; // mar NEM
segmentation fault!

p1.reset();  // Semmit nem csinal
```

unique_ptr

A release() metódus hatására befejezi a mem.terület menedzselését, visszaadja a címét nyers mutatóként, és innentől nullptr címet tárol.

- Ez akkor jó, ha olyan kódnak akarjuk átadni a mem.terület menedzselését, ami nyers pointert vár, vagy egy másik smart pointernek adjuk át.

A get() metódus úgy adja vissza a nyers mutatót, hogy folytatja a mem.terület menedzselését.

- Ez használható, ha egy olyan függvénynek kell átadnunk a címet, ami nyers pointert vár, de nem veszi át a menedzselését. (pl. egy C-ben írt függvény)

shared_ptr

Ha több helyről akarunk hivatkozni egy mem.területre, akkor nem jó a unique_ptr, mert double free hibát okozna, ezért is nem lehet másolni. Erre az esetre való az **std::shared_ptr**.

Le lehet másolni, és számolja, hány másolat létezik (reference counting). Amikor az utolsó másolat is megszűnik (vagy átirányítódik), akkor szabadítja fel a mem.területet. Így nem lesz dangling pointer hiba.

shared_ptr használata

```
std::shared_ptr<int[]> p1(new int[5]);  // 5 egesz szamnak megfelelo
mem.terulet

for (int inx = 0; inx < 5; inx++) { p1[inx] = inx; }

std::shared_ptr<int[]> p2 = p1;  // p1 es p2 is tulajdonosa a mem.teruletnek


p1.reset();  // A mem.terulet meg nincs felszabaditva p2 miatt

//std::cout << p1[2] << std::endl; // segfault!!

std::cout << p2[2] << std::endl;

p2.reset();  // A mem.teruletet felszabaditja - nincs tobb hivatkozas ra
```

weak_ptr

Az **std::weak_ptr** objektum egy shared_ptr másolatát tartalmazhatja anélkül, hogy kihatással lenne a shared_ptr életciklusára. Vagyis a shared_ptr függetlenül a weak_ptr másolatoktól marad fenn / törlődik.

Éppen ezért ahhoz, hogy egy std::weak_ptr-t felhasználjunk, először is ellenőrizni kell, hogy a hozzá tartozó mem.terület még létezik-e!

- Ezt megtehetjük úgy, hogy simán a változóra tesztelünk. Vagy pedig használhatjuk az expired() metódust, mint a következő fólián (ez elegánsabb, mert beszédesebb)

Amikor az std::weak_ptr objektumot felhasználjuk, először meg kell rá hívni a lock() metódust – ez átkonvertálja shared_ptr-ré.

- A lock() metódus használható az érvényesség tesztelésére is:
- `if (std::shared_ptr<int> sp = wp.lock())`

weak_ptr használata

```
std::shared_ptr<int[]> p1(new int[5]);
// 5 egesz szamnak megfelelo mem.terulet

for (int inx = 0; inx < 5; inx++) {
p1[inx] = inx; }

std::weak_ptr<int[]> p2 = p1;  // p2 is
p1 mem.területere hivatkozik, de nincs
kihatassal annak fennmaradasara

// std::cout << p2[2] << std::endl; //
fordito hiba!

std::cout << p2.lock()[2] << std::endl;
// aha! igy mar ok

p1.reset();  // A mem.terulet fel is
szabadult!

//std::cout << p1[2] << std::endl; //
segfault!!

//std::cout << p2.lock()[2] << std::endl;
// segfault!!

if (p2.expired()) {

std::cout << "unfortunately, p2 can no
longer be used" << std::endl;

}

p2.reset();  // Nincs hatasa
```

Smart pointerek

A C++11 és C++14 bevezette az **std::make_shared()** és **std::make_unique()** függvényeket is. Ezek lényegében a new operátort váltják ki exception safe módon. A fenti függvények esetében garantáltan nem lesz mem.szivárgás akkor sem, ha a hívás más része kivételt dob.

Ennél a hívásnál a memóriát lefoglalhatja a program és ez elveszik:

```
f(unique_ptr<T>(new T), function_that_can_throw());
```

Itt viszont garantáltan nem lesz memóriaszivárgás:

```
f(make_unique<T>(), function_that_can_throw());
```

A make_shared pedig egyszerre allokálja a referenciaszámláláshoz használt memóriát és az objektumot, így megspórol egy extra allokációt a new-hoz képest.

Smart pointerek

Mivel a smart pointer is egy pointer, használata teljesen kompatibilis a dinamikus polimorfizmussal:

```
class A {
  public: virtual void introduce() {
    std::cout << "A" << std::endl;
  }
};
class B : public A {
  public: void introduce() override {
    std::cout << "B" << std::endl;
  }
};
```

```
void func(std::unique_ptr<A> avar) { avar->introduce(); }


int main()
{
  std::unique_ptr<A> a1 = std::make_unique<A>();
  std::unique_ptr<B> b1 = std::make_unique<B>();
  func(std::move(a1)); // move nelkul fordito hiba! 2 tulajdonos nem birtokolhatja
  func(std::move(b1)); // hoppa! unique_ptr<B> egyfajta unique_ptr<A>!
```

Körkörös hivatkozások

std::shared_ptr-ek esetén ügyelni kell a körkörös hivatkozások elkerülésére. A ciklusok megtörhetőek pl. std::weak_ptr-ekkel.

Előbb nézzünk egy példát az alapproblémára:

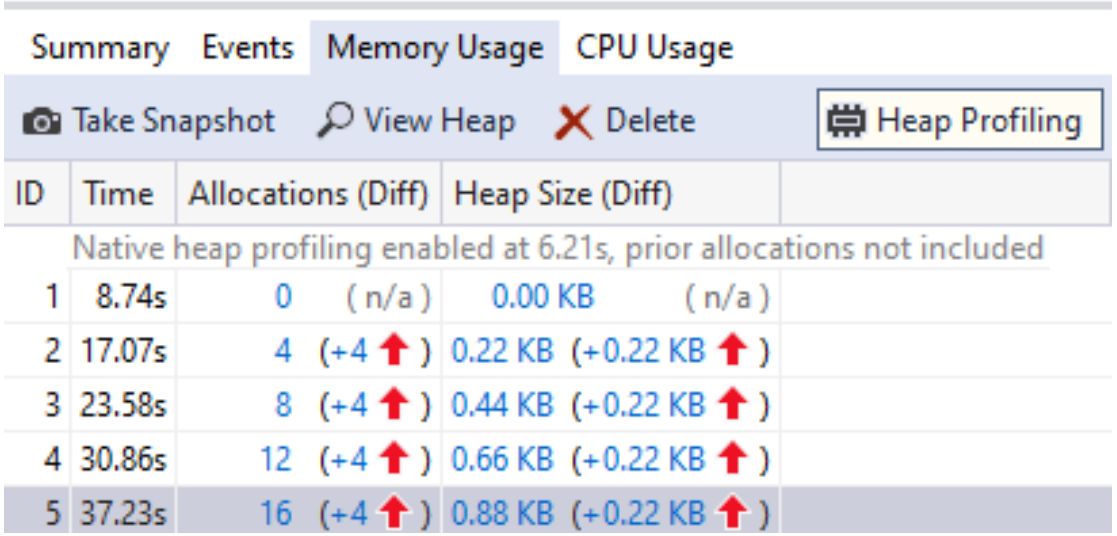
```
struct B;
struct A { std::shared_ptr<B> b;
  ~A() { std::cout << "~A()" << std::endl; }
};
struct B {std::shared_ptr<A> a;
  ~B() { std::cout << "~B()" << std::endl; }
};
```

```
void useAnB() {
  auto a = std::make_shared<A>();
  auto b = std::make_shared<B>();
  a->b = b;
  b->a = a;
}
```

Körkörös hivatkozások

Ha ezt profilozzuk, kiderül, hogy useAnB() lefutását követően nem szabadul fel a memória!

```
int main() {
  char c;
  for (int i = 0; i < 10; i++) {
    useAnB();
    std::cout << "Finished using A and B\n";
    std::cin >> c;
  }
  return 0;
}
```



Körkörös hivatkozások

A megoldás, hogy átgondoljuk: azt szeretnénk, hogy A legyen a “főnök” (aki birtokolja B-t), vagy inkább B birtokolja A-t?

Ha A az elsődleges, akkor B-ben inkább weak_ptr-t hozzunk létre!

```
struct B;

struct A { std::shared_ptr<B> b;

  ~A() { std::cout << "~A()" << std::endl; }

};


struct B {std::weak_ptr<A> a;

  ~B() { std::cout << "~B()" << std::endl; }

};
```

```
void useAnB() {
  auto a = std::make_shared<A>();
  auto b = std::make_shared<B>();
  a->b = b;
  b->a = a;
}
```

Smart pointerek

Mikor melyik smart pointer típust érdemes használni?

Az alapvetően használt típus általában az std::unique_ptr. Ez a típus világossá teszi, hogy aki megkapta, az birtokolja az adott változót / pointert.

- Ha egy függvény unique_ptr-t ad vissza, ez egy move-val egyenértékű. Ilyenkor a hívó tudni fogja, hogy ő birtokolja a mem.címet (szabadon módosíthatja), és a unique_ptr objektum maga fel fogja szabadítani a mem.területet amikor már nincs rá szükség
- Emlékezzünk vissza: unique_ptr típusú objektum nem másolható (csak move-olható)! Persze ha a program rossz felépítésű, előfordulhat, hogy egy másik része eltárolta a unique_ptr-ben levő pointer-t, mint címet. Ekkor 2 helyen is íródhat az érték. De ha ilyen történik, az rossz design... alapvetően a unique_ptr-nek csak 1 tulajdonosa van.

Smart pointerek

Mikor melyik smart pointer típust érdemes használni?

Az alapvetően használt típus általában az std::unique_ptr. Ez a típus világossá teszi, hogy aki megkapta, az birtokolja az adott változót / pointert.

- Ha egy függvénynek meghívásakor (pl. konstruktor) unique_ptr-t kell átadni, világos lesz, hogy az újonnan létrejövő objektum (vagy az adott fv.) birtokolja a mem.területet, és ő is szabadítja fel (amikor megszűnik és a unique_ptr kimegy a scope-ból)
- Ha azt szeretnénk, hogy egy objektumé vagy függvényé legyen egy unique_ptr, de nem szeretnénk, ha módosítani tudná, a becsomagolt pointer mutathat konstans értékre:

```
std::unique_ptr<const Pizza> bakeAPizza();

// itt valamiert nem akarjuk, hogy a Pizzat
modositani lehessen
```

Smart pointerek

weak_ptr-eket leginkább a shared_ptr-ekben levő körkörös hivatkozási láncok megtörésére szokás használni.

std::shared_ptr-re akkor van szükség, ha nincs egyértelmű tulajdonosa a dinamikusan lefoglalt adatnak.

Például egy objektum dinamikusan foglal memóriát, majd lemásoljuk, és a másolattal osztoznak a memórián. Az első objektum megszűnik, de a másik még használja a memóriát, és lehet, hogy további másolatok készülnek belőle. Melyik másolatnak kéne felszabadítania?

A referencia számolás extra overhead-et jelent, és a közös adat elérése nem lesz automatikusan szálbiztos, csak a referenciaszámlálás szálbiztossága garantált.

Fejtörő

Mi a probléma az alábbi kóddal?

```
struct S {
    std::unique_ptr<S> m;
};

int main()
{
    auto p = std::make_unique<S>();
    p->m = std::make_unique<S>();
    std::swap(p, p->m);
}
```