C++ osztályok

OOP és DB - OOP 1. óra

A C++ memória modellje (kis ismétlés)

A lokális scope-ban létrehozott változók a stack-re (verem) kerülnek

- Ha egy függvény visszatér, az általa lefoglalt lokális változók eltűnnek a stack “tetejéről”
- Ha egy függvény meghívódik, a parameter-listában kapott értékekhez létrehoz egy-egy újabb stack változót, ezekbe bemásolja az értékeket és ezek a változók lokális érvényűek

```cpp
#include <iostream>

void someFunction(int a, char b) {
  a = a+1;
  std::cout << "a + b + 1 = " << a + b << std::endl;
}

int main() {
    int a = 5;
    int b = 4;
    someFunction(a, b);
    std::cout << "a in main() = " << a << std::endl;
    return 0;
}
```

```
clang++-7 -pthread -std=c++17 -o main main.cpp
./main
a + b + 1 = 10
a in main() = 5
```

Itt a someFunction()-ban levő a és b változók nem ugyanazok, mint a main() fv a és b változói

Ezek lokális változók amik inicializálásukkor megkapják a main() fv-ben levő a és b változók **értékét**

A C++ memória modellje (részben ismétlés) – ami új, az piros!

Fentiek akkor is igazak, ha someFunction()-nak egy pointer vagy referenciát adunk át. A someFunction()-ban levő pointer vagy referencia egy új változó a stack-en – más kérdés, hogy ezek egy memória címet (pointer esetében), vagy egy másik változóra való hivatkozást tárolnak értékként.

```cpp
#include <iostream>

void someFunction(int* a, int& b) {
  *a = *a+1;
  std::cout << "a + b + 1 = " << *a + b << std::endl;
  std::cout <<"a in someFunction() = " << *a << std::endl;
}

int main() {
    int a = 5;
    int b = 4;
    someFunction(&a, b);
    std::cout << "a in main() = " << a << std::endl;
    return 0;
}
```

```
clang++-7 -pthread -std=c++17 -o main main.cpp
./main
a + b + 1 = 10
a in someFunction() = 6
a in main() = 6
```

Ez a példa tehát csak annyiban más, hogy a someFunction()-ban levő 4. sor megváltoztatja a main() fv-ben levő a értékét is.

someFunction()-ban levő a egy másik változó, de ez a külső a címét tárolja.

A C++ memória modellje (részben ismétlés)

Mi történik, ha azt szeretnénk, hogy mondjuk someFunction() létrehoz egy változót, de annak az értékét someFunction() visszatérése után is olvasni szeretnénk?

```cpp
#include <iostream>

int* someFunction(int a, int b) {
  a = a+1;
  int* sum = new int;
  *sum = a + b;
  std::cout << "a + b + 1 = " << *sum << std::endl;
  std::cout <<"a in someFunction() = " << a << std::endl;
  return sum;
}

int main() {
    int a = 5;
    int b = 4;
    int* c = someFunction(a, b);
    std::cout << "a in main() = " << a << std::endl;
    std::cout << "c in main() = " << *c << std::endl;
    delete c; // ez fontos!
    return 0;
}
```

```
clang++-7 -pthread -std=c++17 -o main main.cpp
./main
a + b + 1 = 10
a in someFunction() = 6
a in main() = 5
c in main() = 10
```

Erre való a dinamikus memória. A dinamikusan (new operátorral) lefoglalt memória nem a stack-en, hanem a heap-en (dinamikus tárhely, free store) fog helyet kapni.

A C++ memória modellje (részben ismétlés)

Az így lefoglalt memória életciklusát a függvények meghívása / visszatérése nem befolyásolja, csak a programozó szabadíthatja fel a delete operator (C-ben free() függvény) segítségével!

```cpp
#include <iostream>

int* someFunction(int a, int b) {
  a = a+1;
  int* sum = new int;
  *sum = a + b;
  std::cout << "a + b + 1 = " << *sum << std::endl;
  std::cout <<"a in someFunction() = " << a << std::endl;
  return sum;
}

int main() {
    int a = 5;
    int b = 4;
    int* c = someFunction(a, b);
    std::cout << "a in main() = " << a << std::endl;
    std::cout << "c in main() = " << *c << std::endl;
    delete c; // ez fontos!
    return 0;
}
```

```
clang++-7 -pthread -std=c++17 -o main main.cpp
./main
a + b + 1 = 10
a in someFunction() = 6
a in main() = 5
c in main() = 10
```

Jóllehet, sum és c itt is lokális változók – a stack-en!

De mivel mindkettő egy pointer (egy mem.cím), az is kérdés, hogy az a cím hol van a memóriában és amit tárol, meddig él.

Erre a címre vonatkoznak a fentiek, hogy a heap-ben van és nem szűnik meg a delete meghívásáig. Ezt jól jegyezzék meg!

A C++ memória modellje (kis ismétlés) – egy későbbi alkalomra

Mindez azt is jelenti, hogy ha someFunction() nem adná vissza a lefoglalt mem.terület címét, bajban lennénk! – hiszen már nem tudnánk elérni a címet (nincsen rá handle-ünk / “fogantyúnk) és a delete-et se hívja már meg semmi.

```cpp
#include <iostream>

void someFunction(int a, int b) {
  a = a+1;
  int* sum = new int; // dangling pointer
  *sum = a + b;
  std::cout << "a + b + 1 = " << *sum << std::endl;
  std::cout <<"a in someFunction() = " << a << std::endl;
}

int main() {
    int a = 5;
    int b = 4;
    someFunction(a, b); // ezutan main()-ben mar nincs mit
    // delete-elni...
    std::cout << "a in main() = " << a << std::endl;
    return 0;
}
```

```
clang++-7 -pthread -std=c++17 -o main main.cpp
./main
a + b + 1 = 10
a in someFunction() = 6
a in main() = 5
```

Az ilyet hívják “dangling pointer”-nek (vagy “dangling reference”-nek referencia esetén)

(A “dangling reference” kicsit máshogy jönne elő – mondjuk someFunction()-ban létrehozunk egy változót, és arra adunk vissza egy referenciát… de amikor visszatér a fv, az a változó már nem él, mivel a stackről eltűnt)

A C++ memória modellje (kis ismétlés) – egy későbbi alkalomra

Dangling reference – ilyet sajnos OOP vizsgán is sokat láttam ☹

Úgy tűnik, mintha someFunction() egy int-et adna vissza, de valójában egy int& (int referenciát) ad vissza! Ez viszont olyan dologra hivatkozik, ami a someFunction() visszatérésével megszűnik:

```cpp
#include <iostream>

int& someFunction(int a, int b) {
  int c = a + b + 1;
  return c;
}

int main() {
    int a = 5;
    int b = 4;
    int c = someFunction(a, b);
    std::cout << "a+b+1 in main() = " << c << std::endl;
    return 0;
}
```

```
clang++-7 -pthread -std=c++17 -o main main.cpp
main.cpp:5:10: warning: reference to stack memory associated with
      local variable 'c' returned [-Wreturn-stack-address]
  return c;
         ^
1 warning generated.
./main
a+b+1 in main() = 10
```

A C++ memória modellje (kis ismétlés) – egy későbbi alkalomra

Tehát: mindig legyünk tudatában annak, hogy mi az ami a stack-en van és mi az ami a heap-en!

Amit egy függvényben nem new-val hozunk létre, az mindig a stack-en van.

```cpp
#include <iostream>

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!
}
```

```
clang++-7 -pthread -std=c++17 -o main main.cpp
./main
9
```

Más kérdés, hogy ha egy összetett objektumról van szó, akkor annak lehetnek részei amik már a heap-en kerülnek allokálásra

Itt például az X osztálynak van egy int* tagváltozója. Ez egy memória címet tároló változó, ami X példányosításakor a stack-en jön létre. Viszont az a memória terület, amire ez az int* hivatkozik, az már a heap-en van!

A C++ memória modellje (kis ismétlés) – egy későbbi alkalomra

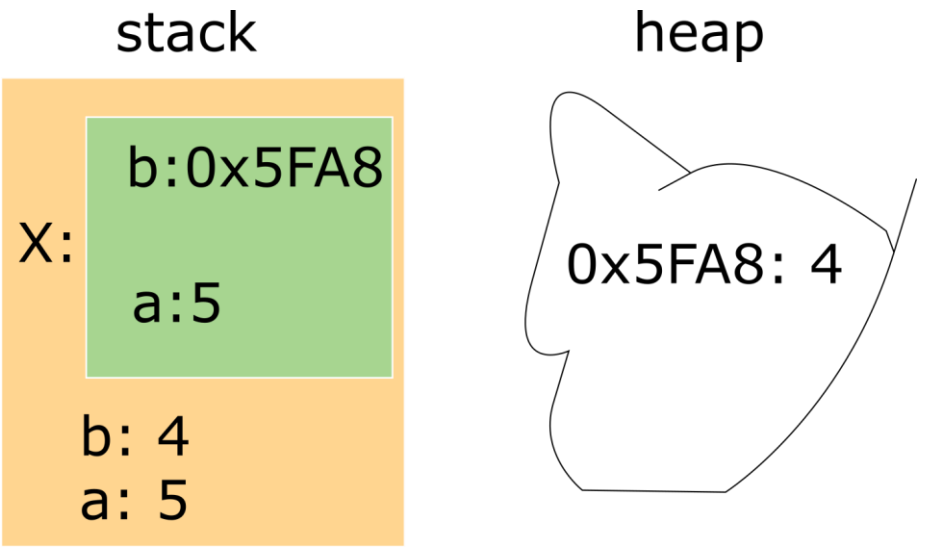
A helyzet valahogy így néz ki:

```cpp
#include <iostream>

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!
}
```

```
clang++-7 -pthread -std=c++17 -o main main.cpp
./main
9
```



A C++ memória modellje (kis ismétlés) – egy későbbi alkalomra

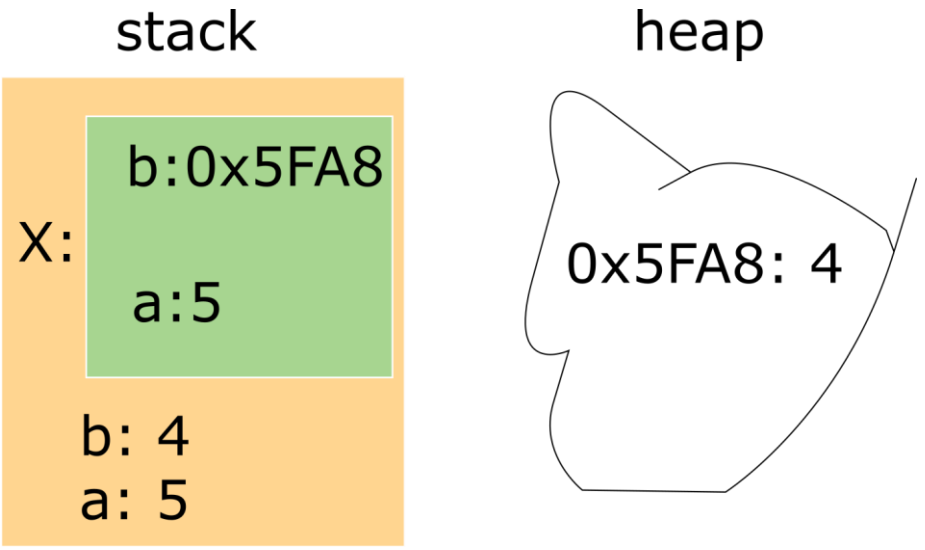
Dangling pointer olyankor tud létrejönni, ha a stack-en egy változó megszűnik, de a hozzá tartozó heap memória nincs felszabadítva. Ezért fontos a destructor, amit meg is írtunk: ha egy X típusú változó a stack-en felszabadul, automatikusan meghívódik a destruktora és felszabadul a heap adott memóriaterülete is.

```cpp
#include <iostream>

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!
}
```

```
clang++-7 -pthread -std=c++17 -o main main.cpp
./main
9
```



Structok

A korábbiakban mindenki megismerhette a C++-ban a struct-ok működését.

A struct egy felhasználó által definiált típus / adatszerkezet, amiben változók és jellemzően hozzájuk tartozó függvények találhatóak. Például:

```
struct Date {
    int d, m, y;
    void init(int dd, int mm, int yy); //inicializalashoz
    void add_year(int n);
    void add_month(int); //signature: nem muszaj nevet adni az arg-nak
    void add_day(int n);
};

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

Structok

A structok legfőbb hátulütője, hogy alapértelmezett esetben nem valósítják meg az adatrejtés elvét.

Minden tagváltozó alapértelmezett esetben publikus, így kívülről is módosítható, ami nehezen követhető / inkonzisztens állapotokhoz vezethet.

```
Date mydate;
mydate.init(5, 12, 1977);
mydate.d = 32; // hupsz, inkonzisztens!
std::cout << mydate.y << "/" << mydate.m << "/" << mydate.d << std::endl;
```

Persze minden további nélkül létre lehet hozni privát tagokat is, de a structok esetében nem ez az alapértelmezett viselkedés.

C++ osztályok

Az osztályok **felhasználó által definiált típusok**, melyek / melyeknek:

- Tagváltozókat és tagfüggvényeket tartalmazhatnak
- Tagfüggvényei definiálják a létrehozás, inicializálás, másolás, mozgatás és törlés működését
- Tagfüggvényei felüldefiniálhatnak operátorokat (+, -, *, /, !, [])
- Tagváltozóik és tagfüggvényeik tekintetében saját névteret alkotnak
- Kívülről elérhető függvényeiket public, implementációjukat private / protected tagokkal valósítják meg. Alapértelmezett esetben az osztály minden tagja privát! (a structokkal ellentétben)

Példa: Date osztály deklarációja

```
class Date
{
    int d, m, y; //default: minden private
public:
    void init(int dd, int mm, int yy);
    void add_year(int n);
    void add_month(int); //signature: nem muszaj nevet adni az arg-nak
    void add_day(int n);
    void print();
};
```

Példa: Date osztály definíciója

Egy típus deklarációja megmondja a fordítónak, hogy példányosításkor mekkora memória-területre van szükség.

Ez alapján akár példányosítható is az objektum / típus (de implementáció nélkül nem feltétlenül használható).

> Ezért van, hogy ha egy .h fájlban van a deklaráció és .cpp fájlban a definíció, akkor egy másik .cpp fájlból (pl. main.cpp) elég a .h fájlt include-olni - a fordító nem fog panaszkodni.
>
> (A teljes használhatóságot majd a linker biztosítja, amikor a külön lefordított .cpp fájlokból keletkező object fájlokat összelinkeli)

Date osztály - metódusok definíciója

A metódusok definiálhatóak az osztályon belül, vagy rajta kívül (egy olyan fájlban, ahonnan a deklaráció látszik). Utóbbi esetben ügyelni kell arra, hogy az osztály saját névteret képez!

```
#include <iostream>

class Date
{
    int d, m, y; //default: minden private
public:
    void init(int dd, int mm, int yy);
    void add_year(int n);
    void add_month(int); //signature: nem muszaj nevet adni az arg-nak
    void add_day(int n);
    void print() { // belul definialt metodus...
        std::cout << "Date is: " << d << "/" << m << "/" << y << std::endl;
    }
};

// kivul definialt metodus:
void Date::init(int dd, int mm, int yy) {
    d = dd;
    m = mm;
    y = yy;
}
```

Date osztály - metódusok definíciója

```
class Date
{
    int d, m, y; //default: minden private
public:
    void init(int dd, int mm, int yy);
    void add_year(int n);
    void add_month(int); //signature: nem muszaj nevet adni az arg-nak
    void add_day(int n);
    void print() { // belul definialt metodus...
        std::cout << "Date is: " << d << "/" << m << "/" << y << std::endl;
    }
};
```

Date osztály - metódusok definíciója

```
// kivul definialt metodus:
void Date::init(int dd, int mm, int yy) {
    d = dd;
    m = mm;
    y = yy;
}

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

void Date::add_month(int n) {
    m = m + n;
}

void Date::add_day(int n) {
    d += n;
}
```

```
int main()
{
    Date mydate;
    mydate.init(23, 7, 1985);
    mydate.print();

    mydate.add_year(1);
    mydate.add_month(3);
    mydate.add_day(2);
    mydate.print();

    std::cin.get();
    return 0;
}
```

Deklaráció és definíció szeparálási elve

Általában célszerű a header file-okba tenni az osztályok deklarációját, a definíciót pedig az osztályon kívül, egy külön fordítási egységbe (.cpp fájlba) tenni. Miért?

A program többi része csak a kisebb méretű header file-t kell, hogy include-olja - garantáltan nem másolódik be több fordítási egységbe u.az az implementáció

Ha egy osztály implementációja megváltozik, elég csak az adott forrás fájlt (.cpp) újra lefordítani, majd linkelni - a program többi részét nem kell újra fordítani.

Mikor lehet érdemes egy metódust mégis az osztály deklarációján belül implementálni?

Abban az esetben, ha nagyon rövid egy metódus törzse, gyakran mégis az osztály deklarációján belül (.h fájlban) szokás definiálni.

A fordító ilyenkor a metódust *inline*-olhatja. Ez azt jelenti, hogy a metódus meghívásának (összes) helyére bemásolhatja a metódus törzsét, így a metódus meghívásakor nem történik context switch (nem jelenik meg a stack-változók menedzselésével járó pluszköltség).

Ez egy lehetőség, de nem minden fordító él vele minden esetben.

Date osztály értékelése

Mi az, ami a jelenlegi Date osztályban jó és mi az ami rossz?

Ami jó, az jó

Pozitívum, hogy:

A Date osztályban megvalósul az adatrejtés

A nap, hónap és év tagváltozók privát elérésűek, vagyis csak az osztály implementációjából(*) érhetőek el

Ha egy dátum valaha is rosszul jelenik meg, arról csakis az osztály metódusai tehetnek (a debuggolás első lépése - a *lokalizáció* - már futtatás előtt megtörténik)

(*) Ez alól kivételt képeznek a friend osztályok és metódusok, amikről később lesz szó.

Ami rossz, azon változtatni kell

Hiányosság, hogy:

(Az implementáció kissé bárgyú, nem ellenőrizzük, hogy az adatok konzisztensek-e, de ezt most hagyjuk)

Az init() metódust nem kötelező meghívni. Mi történik, ha egy programozó (az osztály, mint interfész / API felhasználója) elfelejti?

```
int main()
{
    Date mydate;
    //...
    mydate.print(); // whoops!
```

Konstruktor

Erre a problémára nyújt megoldást a **konstruktor.**

A konstruktor az osztály tagfüggvénye, mely / melynek:

- neve megegyezik az osztály nevével
- nincs visszatérési értéke
- akárhány és akármilyen típusú argumentuma lehet (így egy osztálynak lehet akárhány konstruktora, csak mindegyiknek más számú és / vagy típusú argumentuma kell, hogy legyen)

Konstruktor - automatikus default konstruktor

Ha nem definiálunk egy konstruktort sem, a fordító automatikusan generál egy argumentum nélküli default konstruktort. Ekkor az osztály kinézetre példányosítható úgy, mint egy sima beépített típus - bár több, mint valószínű, hogy nem lesz jól inicializálva!

```
int main()
{
    Date mydate; // mivel nincs konstruktor, automatikusan generalodik egy default konstruktor
    // ugyanakkor ez semmi hasznosat nem fog csinalni
```

Bonyolult szabályok vannak arra, hogy ilyenkor az egyes tagváltozók milyen kezdeti értéket kapnak (hogyan inicializálja őket az automatikusan generált default konstruktor). Pl. függ attól, hogy statikus allokálásúak vagy a heap-en vannak.

Konstruktor - automatikus default konstruktor

Ráadásul ha a tagváltozók között nemcsak alaptípus van, hanem összetett típus is, amit nem lehet üres argumentum-listával inicializálni, az automatikusan generált default konstruktor nem lesz jó - hibát fog eredményezni.

Ezért - annak ellenére, hogy a fordító legenerál nekünk egy default konstruktort - **érdemes mindig saját konstruktort definiálni**!

> Még akkor is, ha a saját konstruktorunk sem vár egyetlen argumentumot sem, esélyes, hogy az általunk definiált konstruktor legalább helyesen tud működni.

Ha egy osztályban legalább 1 konstruktort definiálunk, a fordító már nem fog default konstruktort generálni, és kötelező az egyik létező konstruktort használni!

Konstruktor - automatikus konstruktorok

Halkan megjegyezzük, hogy ha nem definiálunk saját copy konstruktort (amiről később lesz szó), ilyet is automatikusan legenerál a fordító.

A copy konstruktor majd azt a célt szolgálja, hogy egy már létező objektum alapján példányosítsunk és inicializáljunk egy új objektumot.

A példára visszatérve: Egy lehetséges konstruktor

```
class Date
{
    int d, m, y; //default: minden private
public:
    Date(int dd, int mm, int yy);
    void add_year(int n);
    void add_month(int); //signature: nem muszaj nevet adni az arg-nak
    void add_day(int n);
    void print() { // belul definialt metodus...
        std::cout << "Date is: " << d << "/" << m << "/" << y << std::endl;
    }
};
```

Vesd össze:

```
class Date
{
    int d, m, y; //default: minden private
public:
    void init(int dd, int mm, int yy);
    void add_year(int n);
    void add_month(int); //signature: nem muszaj nevet adni az arg-nak
    void add_day(int n);
    void print() { // belul definialt metodus...
        std::cout << "Date is: " << d << "/" << m << "/" << y << std::endl;
    }
};
```

Konstruktor implementációja lehet kívül vagy belül

```
class Date
{
    int d, m, y; //default: minden private
public:
    Date(int dd, int mm, int yy);
    Date() { d = 1; m = 1; y = 1970; } // default konstruktor
    // ...
};

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

Ha vannak konstruktorok, kötelező valamelyiket használni

```
int main()
{
    Date mydate(23, 7, 1985);
    Date mydate2; // ez is ok, mert van default konstruktor
    // ha nem lenne, hibat dobna a fordito
    mydate.print();
    mydate2.print(); // 1/1/1970
```