C++
Gyakorlat jegyzet
2 óra.

A jegyzetet Umann Kristóf készítette Porkoláb Zoltán és Horváth Gábor. (2016. december 27.)

1. Láthatóság, élettartam

Egy objektum **láthatóságának** nevezzük a kódnak azon szakaszait, melyeknél lehet rá hivatkozni.

Egy objektum **élettartamának** nevezzük a kód azon szakaszát, melynél bent szerepel a memóriában. Amikor egy objektum élettartama elkezdődik, azt mondjuk, az objektum létrejön, míg az élettartam végén az objektum megsemmisül.

1.0.1. Megjegyzés. Ez alapján megállapíthatjuk, hogy egy globális változó láthatósága és élettartama a program futásának elejétől végéig tart.

Figyeljük meg, mikor tudunk `x` változóra hivatkozni (azaz hol lesz `x` látható)!

```
int x;

int main()
{
    int x = 1;
    {
        int x = 2;
        std::cout << x << std::endl; // 2
    }
}
```

Megfigyelhető, hogy a `main` függvény elején létrehozott `x` az utána következő blokkban teljesen elérhetetlen – nincs olyan szabványos nyelvi eszköz, amivel tudnánk rá hivatkozni. Ezt a folyamatot **leárnyékolás**nak (*shadowing*) nevezzük. Azonban a külső, globális `x`-re bármikor tudunk hivatkozni az alábbi módon:

```
int x;

int main()
{
    int x = 1;
    {
        int x = 2;
        std::cout << ::x << std::endl; // 0
    }
}
```

2. A stack működése

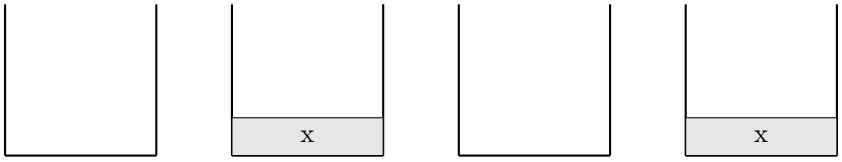
A stack a c++ alapértelmezett „memóriája”, minden változó alapértelmezetten itt jön létre és semmisül meg. Amennyiben egy változóra többet nem tudunk hivatkozni, automatikusan megsemmisül.

```
#include <iostream>

int f()
{
    int x = 0; //x letrejon
    ++x;
    return x;
} //x megsemmisul

int main()
```

```
{
    std::cout << f() << std::endl;
    std::cout << f() << std::endl;
    std::cout << f() << std::endl;
    std::cout << f() << std::endl;
    std::cout << f() << std::endl;
}
```

Kimenet: `1 1 1 1 1`

A fenti kód működését így képzelhetjük el:



Az ábrán egy stack-et látunk. Amikor a vezérlés az `f` függvényhez ér, és ott létrehozza az `x` változót, azt behelyezi a stack-be. A `return` kulcsszó hatására készít `x`-ről egy temporális példányt, ami a függvény visszatérési értéke lesz. Amikor a vezérlés visszatér a `main` függvényhez, x-re nem tudunk tovább hivatkozni, így azt megsemmisíti, és ez ismétlődik folyamatosan.

A stack egy FILO (*first in last out*) adatszerkezet – azaz azt az elemet „dobja” ki a vezérlés a stack-ből, melyet utoljára rakott be.

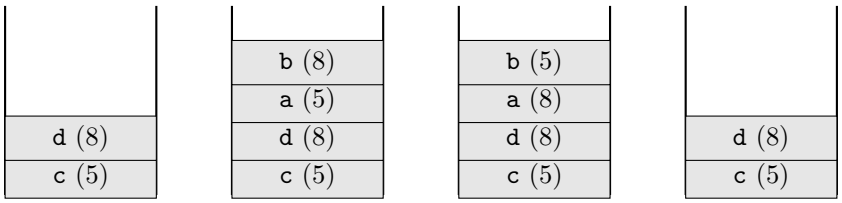
3. Paraméter átvétel

3.1. Érték szerinti paraméter átvétel

Próbáljuk megvalósítani a swap függvényt!

```
#include <iostream>
void swapWrong(int a, int b)
{
    int tmp = a;
    a = b;
    b = a;
}

int main()
{
    int c = 5, d = 8;
    swap(c, d);
    std::cout << c << ' ' << d << std::endl;
}
```

A program kimenete `5 8` ment. Ez egy teljesen jól definiált viselkedés. Ez azért van, mert itt **érték** szerint vettük át (*pass by value*) `a` és `b` változót. A következő ábrán megfigyelhetjük miért is nem. Képzeljük el, ahogy ebbe a verembe a kódunk elrakja a `c` és `d` változókat. Majd meghívja a `swapWrong` függvényt, melyben létrehozott `a` és `b` változókat ismét behelyezi. Bár a függvényre lokális `a` és `b` változókat megcseréli, de a függvényhívás után ezeket ki is törli a stackből.



C++ban alapértelmezett a paraméterátadás függvényeknél érték szerint történik.

3.2. Mutatóval történő paraméter átvétel

A mutatók olyan nyelvi elemek, melyek egy memóriaterületre mutatnak. Segítségükkel anélkül is tudunk hivatkozni egy adott objektumra (és nem csak a másolatára), hogy közvetlenül az objektummal dolgoznánk. Most röviden megismerkedünk velük, de később részletesebben visszatérünk rájuk.

```
int main()
{
    int c = 5, d = 8;
    int *p = &c;
}
```

A fenti példában `p` egy mutató (*pointer*), mely egy `int` típusra mutat. Ahhoz, hogy értéket tudjunk adni egy mutatónak, egy memóriacímet (*referenciát*) kell neki értékül adni, amire rá tud mutatni, erre való a **referáló operátor** (&). Ha a mutató által *mutatott értéket* szeretnénk módosítani, akkor dereferálnunk kell a **dereferáló operátor**ral (*).

```
int *p = &c; //referaljuk c-t
*p = 4; //dereferaljuk p-t
p = &d;
*p = 7;
```

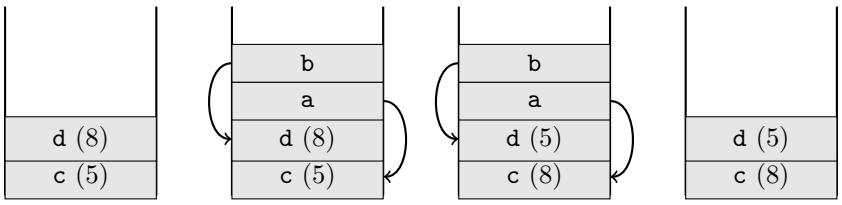
Rendre: pointer inicializálása, pointer által mutatott érték módosítása, pointer átállítása másik memóriacímre, és a mutatott érték módosítása.
Egy mutató mutathat változóra, másik mutatóra, saját magára, és sehova is. Azok a mutatókak, melyek sehová sem mutatnak, nullpointernek nevezzük, és így hozhatjuk létre őket:

```
p = 0;    p = NULL;    p = nullptr;
```

3.2.1. Megjegyzés. Ez a három értékadás (közel) ekvivalens, azonban a `nullptr` kulcsszó csak c++11ben érhető el.

```
void swapP(int *a, int *b)
{
    int tmp = *a;
    *a = *b;
    *b = tmp;
}
```

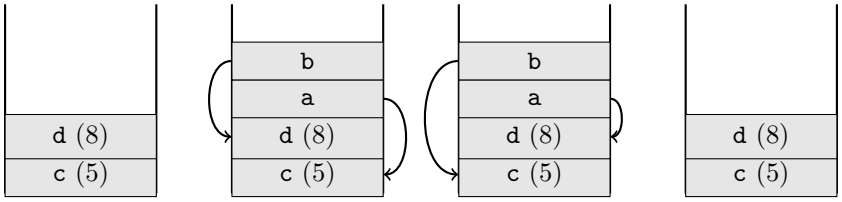
Amennyiben ezt a függvényt hívjuk meg, valóban megcserélődik a két változó értéke. De ehhez fontos, hogy ne simán `swapP(c, d)`-t írjunk függvényhívásként, az ugyanis az fordítási hibához vezetne, mert a `c` és `d` típusa `int`, és nem `int*`. Ahhoz, hogy értéket adjunk egy pointernek, a `c`-hez és `d`-hez tartozó memóriacímeket kell átadni, így a `swapP(&c, &d)` hívás lesz megfelelő.



Ezt azonban ez még mindig **érték szerinti** átadásnak nevezzük, mert most nem konkrét értéket, hanem a memóriacímet másoltuk át.

```
void swapWrong2(int *a, int *b)
{
    int *tmp = a;
    a = b;
    b = tmp;
}
```

Ebben a példában nem a pointerek által mutatott értéket, hanem magukat a pointereket cseréljük meg. Itt annyi fog csupán történni, hogy a függvény beljesében `a` és `b` pointer másra fog mutatni. De annak értéke nem változik.



3.3. Referencia szerinti paraméter átvétel

Megállapíthatjuk, hogy az előző megoldásnál nem változtattuk meg azt, hogy mire mutassanak a pointerek, így azokat konstansként is definiálhatnánk. A konstans pointerek módosíthatják a mutatott cím értékét, de máshova nem mutathatnak. Úgy tudunk egy ilyen pointert létrehozni, hogy a csillag után írjuk a `const` kulcsszót.

```
void swap(int * const a, int * const b)
{
    //...
}
```

Egy kis szintaktikai cukorkával megúszhatjuk azt, hogy folyton kiírjuk a `* const`-ot (lévén ritkán akarjuk megváltoztatni hogy ilyen esetben a pointer hova mutasson). Erre való a referencia szerinti paraméter átváétel (*pass by reference*). A referencia úgy működik, mintha egy konstans pointer lenne.

```
void swapRef(int &a, int &b)
{
    int tmp = a;
    a = b;
    b = tmp;
}
```

Ez a két függvény lényegében ekvivalensek. A különbség a referencia és a pointer között csupán annyi, hogy egy referencia nem lehet null.

3.3.1. Megjegyzés. Ez bár ezt referencia szerinti átvételnek nevezzük, de itt is történik másolás, a memóriacímet ítt is érték szerint vesszük át.

Megjegyzendő, hogy a fenti `swapRef` függvénynek nem kell memóriacímeket átadni, `swapRef(a,b)`-t kell írnunk.

3.3.2. Megjegyzés. Egy referenciát mindig inicializálni kell. Csak úgy mint egy konstanst (különben fordítási hibát kapunk.)

4. Visszatérési érték problémája

Nem primitív (pl. `int`) típusoknál gyakran megeshet, hogy egy adott ípushoz tartozó pointer mérete kisebb, mint magának az objektumé, így megérheti mindentől függetlenül a paramétert referencia szerint átvenni. Ezen felbátorodva mondhatnánk azt is, hogy referenciával is térjünk vissza (a következő példában tekintsünk el attól, hogy `int`-el dolgozunk, bátran képzeljük azt hogy az pl. egy nagyon nagy mátrix)!

```
int& addOne(int &i)
{
    i++;
    return i;
}

int main()
{
    int i = 0;
    int a = addOne(i);
    std::cout << a << std::endl;
}
```

A fenti kóddal semmi gond nincs is. De mi van, ha egy picit módosítunk rajta?

```
int& addOne(int &i)
{
    int ret = ++i;
    return ret;
}
```

A baj máris megvan, amit egy warning is jelezni fog nekünk: olyan objektumra hivatkozó referenciát adunk vissza, amely `addOne`-on belül lokális. Ez azt jelenti, hogy amint a vezérlés visszatér a `main` függvényhez, `ret` megsemmisül, és a `main` függvény pedig a `ret`-hez tartozó címen lévő értéket próbálná meg lemásolni. Mivel viszont a `ret` már ezen a ponton megsemmisült, semmi nem garantálja, hogy azon a memóriaterületen ne következett volna be módosítás.

Az olyan memóriaterületre való hivatkozás, mely nincs a program számára lefoglalva, nem definiált viselkedést eredményez.

5. Kifejezések kiértékelése.

5.0.1. Példa.

```
#include <iostream>

char* answer (char *q);

int main()
{
    std::cout << answer("Hogy␣vagy?") << answer("Biztos?") << std::endl;
    return 0;
}

char* answer (char *q)
{
    std::cout << q;
    static char buffer[80];
    std::cin.getline(buffer,80);
    return buffer;
}
```

Itt már azt is meg akarjuk kérdezni, hogy biztos-e. Itt már találkoztunk a problémával, hogy a kiíratás sorrendje rossz.
`std::cout << answer("Biztos?") << answer("Hogy vagy?") << std::endl;`
Ez már jó. (a kiértékelés nem definiált, de a kiíratási sorrend igen!)
Ez az igazán jó megoldás, itt kevesebbet kell filózni:
std::cout << answer("Hogy vagy?");
`std::cout << answer("Biztos?");`

Azonban a statikus változótól még nem szabadultunk meg. Egy másik megoldás lehet a dinamikus memória kezelés.

A dinamikusan lefoglalt memória az „átlagos” stacken lévő objektumokkal szemben a mi felelőségünk teljesen. Nekünk kell őket allokálni, és ha nincs már rá szükségünk, nekünk is kell felszabadítani. c++11ben smart-pointerekkel ezt valamelyest automatizálhatjuk.

```
#include <iostream>

char* answer (char *q);

int main()
{
    std::cout << answer("Hogy␣vagy?");
    std::cout << answer("Biztos?");
    return 0;
```

```
}

char* answer (char *q)
{
    std::cout << q;
    char* buffer = new char[80];
    std::cin.getline(buffer,80);
    return buffer;
}
```

Ez így nagyon szép megoldás, de a memória sajnos elúszott. Ahogy említve volt, a `new` kulcsszóval létrehoztunk a dinamikus tárhelyen egy új változót, de azt soha nem szabadítottuk fel. Ezért a legszebb megoldás még mindig az, hogyha referenciával átadok még egy paramétert, amiben el tudjuk tárolni a választ. Ennél azonban még egyszerűbb megoldás az, ha az `std::string`-et használjuk.

```
#include <iostream>

std::string answer (char *q);

int main()
{
    std::cout << answer("Hogy␣vagy?");
    std::cout << answer("Biztos?");
    return 0;
}

std::string answer (std::string q)
{
    std::cout << q;
    std::string buffer;
    std::cin >> buffer;
    return buffer;
}
```

Ez a memóriában úgy néz ki, hogy a stacken létrejön egy pointer, heapre (vagy dinamikus tárhelyen) mutató területen tárolja el a buffert, copy konstruktorral adjuk vissza megoldást, a buffer destruktora felszabadítaná a tárhelyet. Ez így igen költséges. c++11ben annyi segítséget kapunk, hogy a move szemantika javít a hatékonyságon

5.0.2. Példa.

```
#include <iostream>

int main()
{
int i = 1;
std::cout << i << ++i << std::endl;
return 0;
}
```

Ez egy **nem definiált viselkedés.** Itt látható egy >> operátor, ami így is felírható: `std::cout.operator<<(i)`. Ennek a függvényhívásnak van visszatérési értéke, méghozzá `std::cout`, így a függvényhívás láncolható. Ez itt egy member function, mellyel majdnem minden alaptípus rendelkezik Ezalól kivétel a `std::string`, melynek `operator>>`-ja globális.

5.0.3. Megjegyzés. Ennek az is lehet értelme, hogy ne függjön az operátor az osztálytól. Jó példa erre a `template`, mert annak csak akkor kell példányt létrehoznia, ha meghívják.

A fenti kódban lévő rész így is felírható:
`std::cout.operator<<(i).operator<<(++i)`.
Az, hogy a második szám 2 lesz, az biztos. De hogy az első mennyi, az nem definiált.
1. ábra.

5.0.4. Példa. `x = k + 2    y = k + 2`

Ebben a példában (jó eséllyel) a fordító kioptimalizálja ezt, és `k+2`-t csak egyszer számolja ki. A c++ban a nem szabványba foglalt szabályoknak köszönhetően sokkal hatékonyabb programokat kaphatunk, mert a fordítónak nagy szabadsága van abban, hogyan optimalizálja a kódunkat.

5.0.5. Példa. Itt a cél az lenne, hogy a tömb elemeit feltöltsük növekvő számokkal.

```
int i = 0;
int t[10];
while (i < 10)
{
t[i] = i++;
}
```

Azonban ez egy nem definiált viselkedés, mert hiába van ott egy **post-fix** `++` operator, az hogy az egyenlőség melyik oldalán levő `i` értékelődik ki először, az ismét nem definiált.

Itt leggyakrabban szekvenciapontok használata tud segíteni.

5.0.6. Definíció. (szekvenciapont) ami elválasztja, hogy mikor minek kell végrehajtódnia futási időben. A szekvenciapont előtt minden kifejezésnek ki kell értékelődnie. Több szekvenciapont létezik: vessző, `&&`, `||`, `?:`

5.0.7. Példa.
```
f(i), ++i;
i++<10 && f(i);
i++<10 || f(i);
i++<10 ? f(i) : g(i);
```

Ezek mint definiáltak, minden kifejezést egy szekvenkciapont választ el a másiktól.

```
f(i++, j++);
```

Itt azonban az, hogy i vagy j értéke növekszik-e meg először, az már nem definiált. Bár valóban található ott vessző, de a vessző mint szekvenciapont nem ekvivalens a függvény paramétereit elválasztó vesszővel.

5.0.8. Megjegyzés. Az optimalizálás nagyon fontos szabálya, hogy mindig úgy szabad csak megtörténnie, hogy a program kimenetele ne változzon.

5.0.9. Megjegyzés. Ha hibásra optimalizálja a kódot a compiler, az nagy szívás. Ez leggyakrabban multi-threaded programoknál fordulhat elő.

```
int f() {cout << 'f'; return 2;}
int g() {cout << 'g'; return 1;}
int h() {cout << 'h'; return 0;}
```

Mi fog történni `f() == g() == h()` kód írásakor?
Itt azon fog múlni a dolog, hogy milyen sorrendben értékelődnek ki az egyenlőség-vizsgáló operátorok. Az operátoroknak van megadott precedenciájuk: erős például a pont, nyíl, [], stb, gyengébb ennél a dereferencia, és így tovább. Azonban az azonos precedenciájú kifejezéseknél kérdéses, milyen sorrendben értékelődnek ki, vagy egyáltaln definiált-e az. Régen fortran-ban ez különösképp problémás volt:

```
A*B / C*D
```

Itt nem lehetett tudni, hogy először megszorozza `A*B`-t, `D`-vel, és csak utána osztja le `C`-vel, vagy fordítva.
Visszatérve a fenti példára, a végrehajtási sorrend: `(f() == g()) == h()`. Azaz, a `==` operátor balról jobbra asszociatív. De milyen sorrendben lesznek kiírva a karakterek? Ez (brace yourselves) nem definiált., hisz az, hogy ezen belül melyik sorrendben fog kiértékelődni a függvényhívás, nincs meghatározva.
Van ahol más a zárójelezés, pl. `!++*++p`. Itt Először előrelépünk a p pointerrel, dereferáljuk, megnöveljük az értékét, és negáljuk. `!(++(*(++p)))`. Ilyen példa szintén az egyenlőség operátor: `x = y = z = 3.14`.

5.0.10. Megjegyzés. Bővebben: `http://en.cppreference.com/w/cpp/language/operator_precedence`

Az optimalizációk azért is segítenek, mert platformspecifikusak gyakran. Úgy csinálja meg a fordítást, hogy az adott gépből a legtöbbet préselje ki.