C++
Gyakorlat jegyzet
2. óra

A jegyzetet Umann Kristóf készítette Horváth Gábor gyakorlata alapján. (2018. április 30.)

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
    }
}
```

1.1. Jobb- és balérték

A láthatóság és élettartam fogalmával szoros összeköttetésben áll a **jobb- és balérték** fogalma. Egy objektumot **balértéknek** (*left value*, röviden *lvalue*) nevezzük, ha címképezhető, és **jobbértéknek** (*right value*, röviden **rvalue**) ha nem. Példaképp:

```
int main()
{
    int *p, r; //balértékek
    &p; //ok, p pointer memóriacímére mutat
    &r; //ok, r memóriacímére mutat
    &5; //nem ok, 5 jobbérték
    &"Hello World!"; //nem ok, "Hello World!" jobbérték
    &&r; //nem ok, címképzés eredménye jobbérték
    5 = r; //nem ok, jobbértéknek nem lehet értéket adni
}
```

A jobbértékek többnyire ideiglenes objektumok, pl. egy érték szerint visszatérő függvény visszatérési értéke, literálok mint pl. `5`, `"Hello World!"`. Lévén ezek az objektumok csak ideiglenesen szerepelnek a memóriában (sőt, gyakran egyáltalán nem, ha a fordító kioptimalizálja, ahogy azt korábban láthattuk), ezért hiba lenne a memóriacímükre hivatkozni, így a fordító ezt nem is engedi.

2. A stack működése

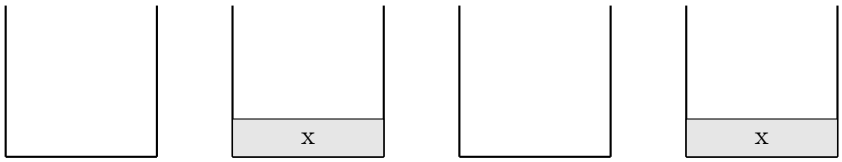
A stack a C++ alapértelmezett tárolási osztálya lokális változók esetén: minden változó alapértelmezetten itt jön létre és semmisül meg. Az itt létrejött változók automatikusan megsemmisülnek. Az élettartamuk a definíciójuktól az adott blokk végéig tart.

```
#include <iostream>

int f()
{
    int x = 0; //x létrejön
    ++x;
    return x;
} //x megsemmisül

int main()
{
    for (int i = 0; i<5; i++)
        std::cout << f() << ' '; // 1 1 1 1 1
}
```

A fenti kód futása során a stack-et így képzelhetjük el:



1. ábra.

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, ameddig a ciklus véget nem ér.

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. Mutatók

A mutatók olyan nyelvi elemek, melyek egy adott típusú memóriaterületre mutatnak. Segítségükkel anélkül is tudunk hivatkozni egy adott objektumra (és nem csak a másolatára), hogy közvetlenül az objektummal dolgoznánk.

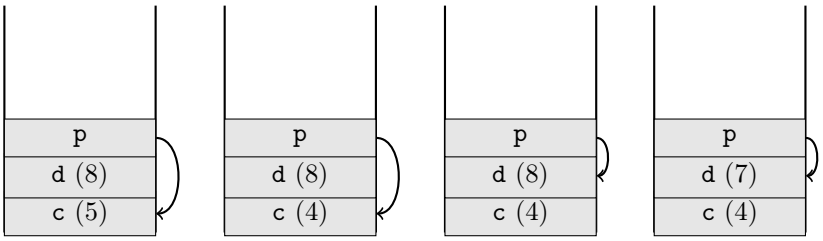
```
int main()
{
    int c = 5, d = 8;
    int *p = &c;
}
```

A fenti példában `p` egy mutató (*pointer*), mely egy `int` típusra mutat. Ahhoz, hogy értéket tudjunk adni egy mutatónak, egy memóriacímet kell neki értékül adni, erre való a **címképző operátor** (&). Ha a

mutató által *mutatott értéket* szeretnénk módosítani, akkor dereferálnunk kell a **dereferáló operátor**ral (*).

A mutatók működését az alábbi példa demonstrálja:

```
int c = 5, d = 8;
int *p = &c; //referáljuk c-t
*p = 4; //dereferáljuk p-t
p = &d;
*p = 7;
```



2. ábra. Az objektum neve mellett zárójelben található az értéke.

Rendre: pointer inicializálása, pointer által mutatott érték módosítása, pointer átállítása másik memóriacímre, és a mutatott érték módosítása.

Egy mutató mutathat változóra, másik mutatóra vagy sehova. Azokat a mutatókat, melyek sehová sem mutatnak, null pointernek nevezzük, és így hozhatjuk létre őket:

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

3.0.1. Megjegyzés. Ez a három értékadás (közel) ekvivalens, azonban a `nullptr` kulcsszó csak C++11ben és azutáni szabványokban érhető el.

3.1. Konstans korrektség

A konstans korrektség egy szabály a C++ nyelvben: ha egy értéket konstansnak jelölünk, azt nem módosíthatjuk a program futása során.

```
const int ci = 6;
int *p = &ci;
```

A fenti kód nem fordul le, mert `ci` konstans, de `p` nem egy nem konstans**ra** mutató mutató, ugyanis ez sérteni a konstans korrektséget. A probléma forrása az, ha fenti értékadás lefordulna, akkor `ci` értékét tudnánk módosítani `p`-n keresztül.

```
const int ci = 6;
const int *p = &ci;
```

A fenti módosítással a kód már lefordul, hiszen a `p` itt már egy konstansra mutató mutató, azaz mutathat konstans változókra. Egy konstansra mutató mutató **nem tudja megváltoztatni** a mutatott értéket, viszont át lehet állítani egy másik memóriacímre.

```
const int ci = 6;
const int *p = &ci;

int c = 5;
p = &c;
```

A fenti kód is szabályos, konstansra mutató mutatóval nem konstans is értékre mutathatunk. Érdemes átgondolni ennek a következményeit, hisz `c` nem konstans, ezért az értékét továbbra is módosíthatjuk (csak nem `p`-n keresztül)! Egy konstansra mutató mutató nem azt jelenti, hogy a mutatott érték sosem változhat meg. Csupán annyit jelent, hogy a mutatott értéket ezen a mutatón keresztül nem lehet megváltoztatni.

```
const int *p = &ci;
int c = 5;
```

```
p = &c;
c = 5;
```

A `const` kulcsszó több helyre is kerülhet.

```
const int *p;
int const *p;
```

A fenti két sor ugyanazt jelenti, a mutatott értéket nem lehet megváltoztatni a mutatón keresztül.

```
int * const p;
```

Amennyiben a * után van a `const`, akkor egy **konstans mutatót kapunk**, mely megváltoztathatja a mutatott értéket, de nem mutathat másra (konstans mutató $\neq$ konstanra mutató mutató). Nyilván, egy pointer lehet egy konstansra mutató konstans mutató is, amin keresztül nem lehet megváltoztatni a mutatott értéket és a mutatót sem lehet máshova átállítani.

```
const int * const p;
```

3.2. Mutatóra mutató mutatók

Mutatóra mutató mutatók is léteznek. Néhány példa:

```
int *p;
int **q = &p;
int ***r = &q;
```

Példaképp `q`-n keresztül meg tudjuk változni, `p` hova mutasson.

```
int c, d;
int *p = &c;
int **q = &p;
*q = &d;
```

A megfelelő szinten a mutatók konstansá tételével még bonyolultabb példákat kaphatunk:

```
int c, d;
int *p = &c;
int * const *q = &p;
*q = &d; // forditasi hiba
```

Mivel `q` egy `int`-re mutató konstans mutatóra mutató mutató, így csak egy olyan mutatóval tudunk rámutatni, ami egy `int`-re mutató konstans mutatóra mutató mutatóra mutató mutató.

```
int c, d;
int *p = &c;
int * const *q = &p;
int *const ** const r = &q;
```

3.2.1. Megjegyzés. Megnyugtatás végett, ritkán van szükség mutatóra mutató mutatónál bonyolultabb szerkezetre.

4. Tömbök

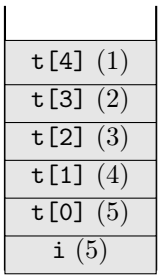
A tömb a C++ egy beépített adatszerkezete, mellyel tömb azonos típusú elemet tárolhatunk és kezelhetünk egységesen. Előzménytárgyakból már megismertük valamennyi funkcionalitását, ám számos veszélyét még nem.

```
int main()
{
    int i = 5;
    int t[] = {5,4,3,2,1};
}
```

`t` egy 5 elemű **tömb**. Nézzük meg, mekkora a mérete (figyelem, ez **implementációfüggő!**)!

```
std::cout << sizeof(i) << std::endl;
std::cout << sizeof(t) << std::endl;
```

A `sizeof` operátor megadja a paraméterként megadott típus, vagy objektum esetében annak típusának méretét (bővebben később). Ez minden implementációra specifikus. Azt látjuk, hogy mindig ötszöröse lesz a `t` az `i`-nek. Azaz a tömbök tiszta adatok. Stacken ábrázolva így képzeljük el:



3. ábra. A `main` függvény változói.

4.1. Biztonsági rések nem definiált viselkedés kihasználásával

Irassuk ki a a tömb elemeit! A példa kedvéért rontsuk el a kódot.

```
for (int i = 0; i < 6; i++) // Hupsz. A csak 5 elem van.
{
    std::cout << t[i] << std::endl;
}
```

Itt látható, hogy túl fogunk indexelni. Ez nem definiált viselkedéshez vezet. Várhatóan memóriaszemetet fog kiolvasni az utolsó elem helyett, de sose tudhatjuk pontosan mi fog történni. Fordítási időben ezt a hibát a fordító nem veszi észre. A gyakorlaton a programot futtatva nem következett be futási idejű hiba.

Most növeljük meg az elemeket, és indexeljünk túl egészen 100ig!

```
for (int i = 0; i < 100; i++)
{
    ++t[i];
}
std::cout << "sajt" << std::endl;
```

Ez a program továbbra is nem definiált viselkedést tartalmaz. Mivel több memóriához nyúlunk hozzá indokolatlanul, ezért nagyobb rá az esély, hogy futási idejű hibába ütközzünk. Az órán a sajt szöveg ki lett írva, mégis kaptunk egy szegmentálási hibát (*segmentation fault*).

```
for (int i = 0; i < 100000; i++)
{
    ++t[i];
}
std::cout << "sajt" << std::endl;
```

A túlindexelést tovább fokozva a program még mielőtt sajt-ot ki tudta volna írni, szegmentálási hibával leállt. Ez jól demonstrálja, hogy ugyanolyan jellegű a hibát követtük el, de mégis más volt a végeredmény. Ez az egyik ok, amiért veszélyesek a nem definiált viselkedések. Mivel számos különböző hibát okozhatnak, ezért a diagnosztizálásuk sem mindig egyszerű. Az alábbi kód szemléltet egy példát, hogyan lehet biztonsági rés a nem definiált viselkedésből.

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

int main()
{
```

```
    int t[] = {5,4,3,2,1};
    int isAdmin = 0;
    std:string name;
    std::cin >> name;
    for (int i = 0; i < name.size(); ++i)
    {
        t[i] = 1;
    }
    if (name == "pityu")
        isAdmin = 1;
    std::cout << "Admin?: " << (isAdmin != 0 ) << std::endl;
}
```

Ha a programnak `pityu`-t adunk meg amikor be akarja olvasni `name`-et, akkor minden rendben. De mivel a forráskódot ismerjük, azért ha hosszú nevet adnánk (nagyobb mint 5), akkor a túlindexelés miatt ki tudjuk használni a nem definiált viselkedéseket. Az is előfordulhat, hogy az `isAdmin` memóriacímére írunk, és elérjük, hogy a szoftver adminként authentikáljon valakit, aki nem az.

Hogyan lehet ezeket a hibákat elkerülni? Túl azon, hogy figyelni kell, vannak programok amik segítenek. Ehhez használhatunk `sanitizer`-eket. Ezek módosítanak a fordító által generált kódon. Létrehoz ellenőrzéseket, amik azelőtt észrevesznek bizonyos nem definiált viselkedéseket, mielőtt azok megtörténnének. Pl. itt a túlindexelés, egy futási idejű hibához és egy jól olvasható hibaüzenethez vezetne. Használatukhoz elég egy extra paranccsal fordítanunk:

```
g++ main.cpp -fsanitize=address
```

A sanitizerek csa abban az esetben találnak meg egy hibát, ha a probléma előfordul (azaz futási időben, nem fordítási időben ellenőriz). Amennyiben előfordul, akkor elég pontos leírást tudunk kapni arról, hogy merre van a probléma. Fordítási időben a figyelmeztetések használata segíthet bizonyos hibák elkerülésében.

```
g++ main.cpp -Wall -Wextra
```

A fenti két kapcsoló szintén extra ellenőrzéseket vezet be, de nem változtatják meg a generált kódot.

4.2. Hivatkozás tömb elemeire

Egy tömb adott elemére több módon is hivatkozhatunk:

```
*(p + 3) == *(3 + p) == p[3] == 3[p]
```

```
#include <iostream>

int main()
{
    int t[][3] = {{1,2,3},{4,5,6}};
    return 0;
}
```

Tekintsük a fenti két dimenziós tömböt. Az első `[]` jelek közt nincs méret, mert a fordító az inicializáció alapján meg tudja állapítani. A második dimenzió méretének megadása viszont kötelező.

Fentebb a tömböknél megadott ekvivalenciát a mátrixra alkalmazva számos indexelési módot le tudunk vezetni:

```
t[1][] == *(*(t+1)+0) == *(1[t]+0) == 0[1[t]] == 0[*(t+1)] == *(t+1)[0] == 1[t][0]
```

4.2.1. Megjegyzés. Előzmény tárgyakból elképzelhető, hogy azt tanultuk, hogy egy tömb méretének egy változót is megadhatunk. Ezt a `gcc` fordító elfogadja és jó kódot is generál belőle. De ez egy nem szabványos kiterjesztés, ezért nem garantált, hogy ezt minden fordító megteszi. Ez jól demonstrálja, hogy a fordítók nem mindenben követik szorosan a szabványt.

5. Paraméter átvétel, visszatérési érték

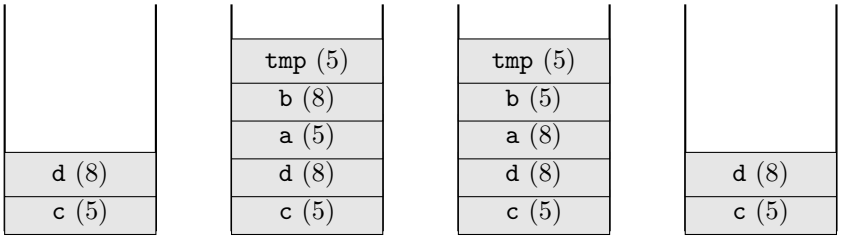
5.1. Érték szerinti paraméter átvétel

C++ban alapértelmezett módon a paraméterátadás érték szerint történik. Figyeljük meg ennek a következményét a `swap` függvény megvalósításával!

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

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

Megfigyelhető, hogy nem sikerült `c` és `d` értékét megcserélni. Ez azonban egy teljesen jól definiált viselkedés. Ennek az az oka, hogy itt **érték** szerint vettük át (*pass by value*) `a` és `b` paramétert. A következő ábrán megfigyelhetjük mi is történik pontosan. Képzeljük el, hogy a stackbe a program elrakja a `c` és `d` változókat. Eztán meghívja a `swapWrong` függvényt, melyben létrehozott `a` és `b` paraméterek szintén a stackre kerülnek. Bár a függvényre lokális `a` és `b` paraméterek értékét megcseréli, de a függvényhívás után ezeket ki is törli a stackből. Az eredeti `c` és `d` változók éréke nem változott a függvényhívás során. (ld.: 4. ábra)



4. ábra. A `swapWrong` függvény szemléltetése a stack-en.

5.2. Mutatók érték szerinti átadása

Nézzük meg, hogy hogyan tudunk megcserélni két értéket ezúttal helyesen, mutatók segítségével.

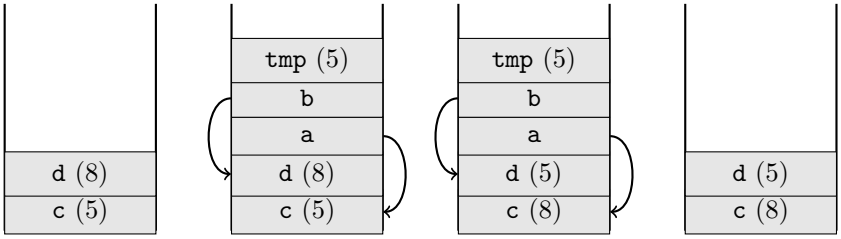
```
void swapP(int *a, int *b)
{
    int tmp = *a;
    *a = *b;
    *b = tmp;
}
```

Amennyiben ezt a függvényt hívjuk meg, valóban megcserélődik a két változó értéke. De ehhez fontos, hogy ne `swapP(c, d)`-t írjunk, az ugyanis az fordítási hibához vezetne, hiszen 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!

```
//...
int main()
{
    int c = 5, d = 8;
```

```
    swapP(&c, &d);
    std::cout << c << ' ' << d << std::endl; // 8 5
}
```

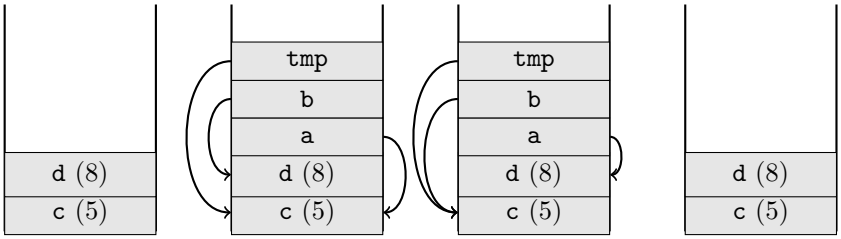
5.2.1. Megjegyzés. A mutatókat továbbra is érték szerint adjuk át. Az `a` és `b` paraméterekben lévő memóriacím tehát a másolata annak, amit a hívás helyén megadtunk.



5. ábra. A `swapP` függvény szemléltetése.

```
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 az fog történni, hogy a függvény belsejében `a` és `b` pointer másra fog mutatni. A mutatott értékek viszont nem változnak.



6. ábra. A `swapWrong2` függvén szemléltetése.

5.3. Referencia szerinti paraméter átadás

Megállapíthatjuk, hogy az előző megoldásnál nem változtattuk meg, hogy mire mutassanak a pointerek, így azokat konstansként is definiálhatnánk. A konstans pointerek módosíthatják a mutatott értéket, de nem lehet őket átállítani egy másik memória címre. 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 nem akarjuk megváltoztatni, hogy ilyen esetben a pointer hova mutasson). Erre való a referencia szerinti paraméter átvaétel (*pass by reference*). A referencia hasonlóan működik, mintha egy konstans pointer lenne, csak nem lehet sehova se mutató referenciát létrehozni.

```
void swapRef(int &a, int &b)
{
```

```
    int tmp = a;
    a = b;
    b = tmp;
}
```

Ez a függvény lényegében ekvivalens a `swapP` fügvénnyel.

5.3.1. 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` függvény meghívásakor nem kell jeleznünk, hogy memóriacímeket akarunk átadni, `swapRef(a,b)`-t kell írnunk.

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

5.4. Visszatérési érték problémája

Nem primitív (pl. `int`) típusoknál gyakran megeshet, hogy egy adott típushoz 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.

5.4.1. Megjegyzés. Értelemszerűen pointerekkel ez ugyanúgy probléma.

5.5. Függvények átadása paraméterként

C++ban lehetőségünk van arra is, hogy függvényeket adjunk át paraméterként. Azokat az objektumokat függvény mutatónak (*function pointer*) hívjuk.

```
int add(int a, int b)
{
    return a + b;
}

int mul(int a, int b)
{
    return a * b;
}

int reduce(int *start, int size, int initial, int (*op)(int, int))
{
    int ret = initial;
    for (int i = 0; i < size; i++)
    {
        ret = (*op)(ret, start[i]);
    }
    return ret;
}

int main()
{
    int t[] = {1,2,3,4,5};
    std::cout << reduce(t,5,0,&add) << std::endl;
    std::cout << reduce(t,5,0,&mul) << std::endl;
}
```

Itt `reduce` egy olyan paramétert is vár, mely igazából egy függvény, amely `int`-et ad vissza, és két `int`-et vár paraméterül.

5.5.1. Megjegyzés. A szavakba öntés segíthet a megértésben: `op` egy olyan függvényre mutató mutató, melynek két `int` paramétere van, és `int` a visszatérési értéke.

A kódban feltűnhet, hogy a tömb mellé paraméterben elkértük annak méretét is. Ennek az az oka, hogy a `t` tömb egy `int`-re mutató mutatóvá fog konvertálódni a paraméter átadás során, ami a tömb első elemére mutat. Ennek hatására elvesztjük azt az információt, hogy mekkora volt a tömb (a tömbök és paraméterátadás kapcsolatról később bővebben lesz szó). Így át kell adni ezt az információt is.

Mellékesen, egy függvény átadásakor csak függvénypointert tudunk átadni. Egy függvénypointeren a függvény meghívása az egyetlen értelmes művelet. Így a `&` jel elhagyható függvényhíváskor és az `op` elől is elhagyható a * a paramétereknél.

```
int reduce(int *start, int size, int initial, int op(int, int)))
{
    //...
}

int main()
{
    int t[] = {1,2,3,4,5};
    std::cout << reduce(t,5,0,add) << std::endl;
    std::cout << reduce(t,5,0,mul) << std::endl;
}
```

5.6. Tömbök átadása függvényparaméterként

Próbáljunk meg egy tömböt érték szerint átadni egy függvénynek!

```
#include <iostream>
```

```
void f(int t[])
{
    std::cout << sizeof(t) << std::endl;
}

int main()
{
    int t[] = {1,2,3,4,5};
    std::cout << sizeof(t) << std::endl;
    f(t);
}
```

Kimenet: `20 8` (implementáció függő)

Bár azt hihettük, hogy `t` tömb méretét írattuk ki ket alkalommal, valójában amikor azt érték szerint próbálunk meg átadni egy tömböt, az átkonvertálódik a tömb elejére mutató pointerré.

```
void f(int t[8])
{
    std::cout << sizeof(t) << std::endl;
}
```

Hiába adunk meg egy méretet a tömbnek a függvény fejlécében, még mindig egy pointer mérete lesz a második kiírt szám. Az a tanulság, hogy ha érték szerint akarunk átadni egy tömböt, az át fog konvertálódni pointerré. A legszebb az lenne, ha a fenti szintaxis nem fordulna le. Ennek azonban történelmi oka van, a C-vel való visszafelé kompatibilitás miatt fordul le.

5.6.1. Megjegyzés. Tömböt értékül adni a szabvány szerint nem is lehet: `int *t2[5] = t` nem helyes.

Korábban megismerkedtünk egy módszerrel, mely segítségével egy tömb méretét (elemszámát) paraméterátadás után is megőriztük:

```
#include <iostream>

void f(int *t, int size) // új paraméter!
{
    std::cout << sizeof(t) << std::endl;
}

int main()
{
    int t[] = {1,2,3,4,5};
    std::cout << sizeof(t) << std::endl;
    f(t, sizeof(t)/sizeof(t[0]));
}
```

5.6.2. Megjegyzés. Amennyiben C++11ben programozunk, érdemes az `std::array`-t használnunk, ami olyan, mint egy tömb, de nem tud pointerré konvertálódni és mindig tudja a méretét.

Ha szeretnénk egy tömböt egy darab paraméterként átadni, megpróbálhatunk egy tömbre mutató pointert létrehozni. Azonban figyelni kell a szintaktikára, ha `int *t[5]`-t írunk, egy öt elemű intre mutató pointereket tároló tömböt kapunk.

Ha tömbre mutató mutatót szeretnék, így csinálhatjuk:

```
void g(int (*t)[5])
{
    std::cout << sizeof(t) << std::endl;
}
```

Azonban ez még mindig egy pointer méretét fogja kiírni, mert a `t` az egy sima mutató! Ahhoz, hogy megkapjuk, mire mutat, dereferálnunk kell, így a `sizeof` paraméterének `*t`-t kell megadni, ha a tömb méretére vagyunk kíváncsiak.

5.6.3. Megjegyzés. Ha refenreciával vennénk át `t`-t, az is hasonlóan nézne ki: `int (&t)[5]`.

Ha eltérő méretű tömböt próbálunk meg átadni, akkor nem fordul le a kód, mert nem egy 5 elemű tömbre mutató mutató 6 elemű tömbre mutató mutatóvá konvertálódni.

```
int main()
{
    int a[6];
    g(&a); //forditasi hiba!
    int b[5];
    g(&b); //ok
}
```

5.6.4. Megjegyzés. Ahhoz, hogy egy olyan függvényt írjunk, ami minden méretű tömböt elfogad paraméterül, a legegyszerűbb megoldás, ha hagyjuk, hogy a tömb átkonvertálódjon egy első elemre mutató pointerré, és átadjuk külön paraméterben a tömb méretét. Bár van megoldás arra is, hogy egy darab "rugalmas" függvényt írjunk, és az egész tömbről csak egy paramétert vegyünk át. Majd a 7-8. gyakorlaton lesz részletesen szó, de a következő a szintaxis:

```
template <class T, int ArraySize>
void ( T (&param)[ArraySize] )
{
    //...
}
```

A működésének az elvét még nem baj, ha nem értjük. A háttérben egy template paraméter dedukció fog végbemenni: a fordító kitalálja `param` méretét.

A tömbök átvétele paraméterként azért ilyen körülményes, mert egy tömbnek a méretét fordítási időben ismernünk kell. Ha változó méretű tömböt várnánk paraméterül, az szembemenne ezzel a követelménnyel.

Könnyű azt hinni (hibásan), hogy a pointerek ekvivalensek a tömbökkel.

```
#include <iostream>

int main()
{
    int t[] = {5,4,3,2,1};
    int *p = t;
    std::cout << *p << std::endl; // 5
    std::cout << sizeof(int) << std::endl; // implementációfüggő, legyen x
    std::cout << sizeof(p) << std::endl; // implementációfüggő, legyen y
    std::cout << sizeof(t) << std::endl; // 5*x
}
```

A fenti program jól szemlélteti, hogy ez nem igaz. A tömb típusa tartalmazza azt az információt, hogy hány elemű a tömb. Egy pointer típusa csak azt az információt tartalmazza, hogy a mutatott elem mekkora. Számos más különbség is van. A tévhit oka az, hogy tömb könnyen konvertálódik első elemre mutató pointerré.