C++
Gyakorlat jegyzet
3 óra.

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

1. Pointer aritmetika

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

Ez nem fordul le, mert `ci` konstans, de `p` nem egy nem konstans pointert. Ez sértené a c++ban ismert **konstans korrektséget** (const correctness). Itt a probléma az lenne, hogy ha rá tudnénk mutatni, akkor hiába lenne `ci` konstans, tudnánk módosítani `p`-n keresztül.

```
const int ci = 6;
const int *p = &ci;
std::cout << *p << std::endl;
```

Ez már jó lesz, mert a `p` egy konstansra mutató pointer, azaz tud mutatni olyan változókra, melyek konstansok. Egy konstansra mutató pointer **nem tudja megváltoztatni** a mutatott memóriacím értékét. Viszont egy konstansra mutató pointer még tud más memóriacímekre mutatni.

```
const int ci = 6;
const int *p = &ci;
std::cout << *p << std::endl;

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

Ez teljesen szabályos, konstansra mutató pointerrel nem konstans értékre mutatunk. Viszont figyelem, `c` nem konstans, azt továbbra is tudjuk módosítani (Csak nem `p`-n keresztül)! Ez meglepetést okozhat, hogyha egy konstans pointer kezelése közben (mely által mutatott terület értékétől nem várnánk hogy változzon) a mutatott cím értéke megváltozik.

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

Szintaktikailag a *-ot sok helyre írhatjuk.

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

Egy kettő ugyanaz, mint fentebb láthattuk.

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

Amennyiben a * után van a `const`, akkor egy **konstans pointert kapunk**, mely megváltoztathatja a mutatott értéket, de nem mutathat másra. (konstans pointer $\neq$ konstanra mutató pointer) Mutatóra mutató mutatók is léteznek.

```
const int **pp = &p;
const int *** const cppp = &pp;
int *const ** const dsfdsf = NULL;
```

Legalul egy integerre mutató const mutatóra mutató mutatóra mutató konstans mutatót látunk (ebben nem vagyok biztos hogy jól írtam le, gondolom nem meglepő miért).

1.0.1. Megjegyzés. Mutatóra mutató mutató (**) még előfordul, de komplikáltabb ritkán.

1.1. Függvény pointerek

C++ban lehetőségünk van arra is, hogy függvényeket adjunk át paraméternek.

```
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, mely `int`-et ad vissza, és két `int`-et vár paraméterül.

A kódban feltűnhet, hogy a tömb mellé paraméterben elkértük annak méretét is. Ez azért van, mert a `t` tömb egy `int`-re mutató mutatóvá fog konvertálni, ami az első elemre mutat. Ennek hatására értelemszerűen elvesztjük azt az információt, hogy mekkora volt a tömb. Így át kell adni ezt az információt is. Mellékesen, függvényeket kezelni csak pointerekkel lehet, és mivel a fordító tudja hogy függvényeket akarunk átadni, 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;
}
```

2. Tömbök

A tömbök a c++ alapértelmezett konténere, mellyel egyszerre tömb azonos típusú elemet kezelhetünk. Előzménytárgyakból már megismertük valamennyi funkcionalitását, ám számos veszélyét még nem.

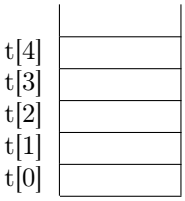
```
#include <iostream>
int main()
{
```

```
    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;
```

Azt látjuk, hogy mindig ötszöröse lesz a `t` az `i`-nek. Azaz a tömbök tiszta adatok. Stacken ábrázolja így képzeljük el:



Irassuk ki a a tömb elemeit! (de ezt basszuk is el!)

```
for (int i = 0; i < 6; i++) //nem 6 elemes
{
    std::cout << t[i] << std::endl;
}
```

Itt előre látható, hogy túl fogunk indexelni. Ez így egy nem definiált viselkedéshez vezet. Várhatóan valamilyen random memóriaszemetet fog kiolvasni (vagy olvashat ki, lévén nem definiált), és sose tudhatjuk pontosan mit. Most növeljük meg az elemeket, és menjünk el egészen 100ig!

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

Ez szintén nem definiált. Mivel olyan memóriaterületeket szeretnénk módosítani, melyeket nem foglaltunk le a programunknak, bajba juthatunk. Itt az órán a sajt szöveg ki lehet írva, mégis kaptunk szegmentálási hiba (*segmentation fault*) hibaüzenetet az oprendszertől.

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

Itt már (legalábbis ebben az esetben) előbb vágta magát hanyat a program, mielőtt sajt-ot ki tudta volna íratni. Ez jól demonstrálja, hogy ugyanazt a hibát követtük el, de más volt a végeredmény. Ezért igazán veszélyesek a nem definiált viselkedések.

```
#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 a legnagyobb rendben. De mivel a forráskódot ismerjük, azért hogyha nagyon hosszú nevet adnánk (nagyobb mint 5), akkor a túlindexelés miatt ki tudjuk használni a nem definiált viselkedéseket, és az is előfordulhat, hogy az `isAdmin` memóriacímére írunk, és elérjük hogy akkor is adminnak higgyen minket, ha nem vagyunk azok.

Hogyan lehet ezeket a hibákat elkerülni? Túl azon, hogy nagyon figyelni kell, vannak programok amik segítenek nekünk. Ehhez használhatunk `sanitizer`-eket. Ezek picit módosítanak a kódunkon, és amennyiben futási időben bizonyos nem definiált viselkedéseket követne el, pl. itt a túlindexelés, leütné a programunkat. Használatukhoz elég egy extra paranccsal fordítanunk:

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

De sajnos ez is csak akkor tud segíteni, ha a probléma előfordul (azaz futási időben, nem fordítási időben ellenőriz). Amennyiben előfordul viszont, elég pontos leírást tudunk kapni arról, hogy merre van a probléma.

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

Ez a 2 parancs szintén extra ellenőrzéseket vezet be, de nem változtatják meg a kódot, csak fordítási időben ellenőriznek.

2.1. Hivatkozás tömb elemeire

```
#include <iostream>

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

Könnyű azt hinni (hibásan) hogy a pointerek ugyanazok mint a tömbök. Ez program jól mutatja, hogy ez nem igaz, mert a tömb tárolja annak méretét is. Számos más különbség is van, viszont egy tömb könnyen konvertálódik pointerré.
Egy tömb adott elemét sokféleképpen le tudjuk kérdezni:

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

```
#include <iostream>

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

Ez egy alternatív módja egy tömb inicializálásának. Itt több dolog megfigyelendő: Az első `[]` operátorban nincs méret, mert a fordító az inicializáció alapján meg tudja állapítani, hogy a mátrix azon dimenziója mekkora, de annyira már nem okos, hogy a másodikat is abszolválja.

Fontos megjegyezni, hogy a mátrix egy adott elemére még többféleképpen tudunk hivatkozni:

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

2.1.1. Megjegyzés. Ahhoz, hogy egy olyan függvényt írjunk ami minden méretű tömböt elfogad paraméterül, a legegyszerűbb megoldás, ha hagyjuk, hogy a tömb átkonvertáljon egy olyan pointerré, ami az első elemre mutat, és átadjuk külön paraméterben a tömb méretét. Bár van megoldás arra is, hogy egy darab "rugalmas" függvényt írjunk, és az egész tömbhöz csak 1 paramétert vegyünk át, annak is komoly hátulütői lehetnek. Majd template-ekkel a 7-8. gyakorlaton lesz részletesen szó, de kb így nézne ki:

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

```
{
    //...
}
```

Ez később jobban ki lesz fejtve, de itt egy template paraméter dedukció fog létrejönni, és a fordító kitalálja `param` méretét. Csak nyilván, mindig amikor egy más méretű tömböt hozunk létre, a fordító példányosítja ezt a függvényt, ami csúnyán meg tudja dobni a bináris kódot (*binary code*).

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.

2.1.2. Megjegyzés. Progalapon úgy tanultunk tömböket, hogy változó méretet adtunk meg nekik. Ezt a `gcc` compiler elfogadja, lefordítja, és jó kódot is csinál belőle, de nem garantált, hogy ezt minden fordító megteszi, ugyanis a c++ szabvány azt mondja ki, hogy a tömb méretének fordítási időben ismertnek kell lennie. Ez jól demonstrálja, hogy a compilerek nem feltételnül követik szorosan a szabványt.