Mutatók (*pointers*)

Olyan nyelvi elemek, melyek egy adott típusú memóriaterületre mutatnak. Segítségükkel anélkül is tudunk hivatkozni egy adott objektumra (nem csak a masolatára!), hogy közvetlenül az objektummal dolgoznánk.

```
int main()
{
  int x = 5;
  int* xPtr = &x;
}
```

A fenti példában `xPtr` egy mutató, amely egy `int` típusra mutat. Ahhoz, hogy értéket tudjunk adni egy mutatónak, egy memóriacímet kell értékül adni, ezt a **címképző operátor** (`&`) segítségével tehetjük meg. Ha a mutató által *mutatott értéket* szeretnénk módosítani, akkor dereferálnunk kell mutatót a **dereferáló operátor** (`*`) segítségével.

```
int main()
{
  int x = 7;
  int* xPtr = &x; // referájuk x-et
  *xPtr = 1; // dereferáljuk xPtr-t, majd módosítjuk a memóriaterületet,
ahová mutat
             // x változó érétke ettől kezdve 1 lesz
}
```

Egy mutató mutathat változóra, másik mutatóra, vagy sehova. Azokat a mutatókat amelyek sehova nem mutatnak **null pointernek** nevezzük. Következő képpen hozhatjuk létre őket:

```
int main()
{
  int* ptr1 = 0;
  int* ptr2 = NULL;
```

```
  int* ptr3 = = nullptr;
}
```

Megjegyzés: a `nullptr` kulcsszót csak a C++11-es szabvány vezette be, így az előtte lévő fordítók (C++98, C++03) nem ismerik fel ezt az operátort. Bevezetésének oka nagyon egyszerű. A `0` vagy a `NULL` típusok egész számmá konvertálódnak ez `NULL` esetén egy 0 lesz. Tekintsük a következő kódrészletet:

```
#include <iostream>

void f(char* b) { std::cout << "char" << std::endl; }

void f(bool i) { std::cout << "bool" << std::endl; }

int main()
{
  f(NULL);
}
```

> Kimenet: *error: call of overloaded 'f(NULL)' is ambiguous.*

```
#include <iostream>

void f(char* b) { std::cout << "char" << std::endl; }

void f(bool i) { std::cout << "bool" << std::endl; }

int main()
{
  f(nullptr);
}
```

> Kimenet: "char".

`nullptr` esetén explicit megtudja határozni a fordító, hogy azt a függvényt kell meghívni amely bemenetként egy mutató típust vár.

Mutatóra mutató mutató

```
int main()
{
  int i = 1;
  int* iPtr = &i;
  int** piPtr = &iPtr; // mutatóra mutató mutató
}
```

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

```
int main()
{
  int i, j;
  int* p = &i;
  int** q = &p;
  *q = &j;
}
```

```
int main()
{
  int i, j;
  int* p = &i;
  int* const * q = &p;
  *q = &j; // fordítási idejű 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 konstans mutató.

```
int main()
{
  int i, j;
  int* p = &i;
  int* const* q = &p;
  int* const **const r = &q;
}
```

Referenciák

A referencia egy létező objektum alternatív neve. Definiálásakor meg kell adni azt az objektumot is, amelyet alternatív névvel látunk el. A referencia nem egy változó, mint a mutató, hanem csak egy azonosító, ezért nem is változtatható meg, amíg a referencia létezik mindig ugyanoda referál. Már létrehozásakor értéket kell adnunk neki, ami a program futása során nem változhat. Két leggyakoribb felhasználása:

- függvény bemeneti paraméter
- függvény visszatérési érték

Mindkét esetben lehet konstans és nem konstans.

```
int main()
{
  int i = 1;
```

```
  int& rI = i; // rI változó egy alias az i változóra
}
```

Konstans korrektség (*const-correctness*)

Mint azt már korábban említettük a **konstans korrektség** egy szabály a C++ nyelvben: ha egy értéket konstansnak jelölünk, azt nem módosíthatjuk a program futása során.

```
int main()
{
 const int i = 1;
 int* iPtr = &i;
}
```

A fenti kód fordítási idejű hibát fog adni. Miért lehet ez? A válasz az, hogy mivel `i` változót konstansnak deklaráltuk viszont `iPtr` nem egy konstansra mutató mutató, azaz ha `iPtr`-el hozzáférnénk `i` memóriacíméhez, akkor a mutatón keresztül módosítani tudnánk a mutatott értéket (ami jelen esetben konstans), ergo sérülne a konstans korreketésg.

Megoldás: deklaráljuk `iPtr` mutatót, hogy konstans értékre mutasson:

```
int main()
{
  const int i = 1;
  const int* iPtr = &i;
}
```

`iPtr` ettől kezdve egy konstanra mutató mutató, így már lehetőségünk van, hogy rámutassunk vele konstans adattagokra is. Konstanra mutató mutatón keresztül bár elérjük a mutatott értéket, de megváltoztatni nem tudjuk.

Annak függvényében, hogy a `*` jobb vagy bal oldalán használjuk a `const` kulcsszót megkülönböztetünk **konstans** mutatót, illetve **konstansra** mutató mutatót.

Konstansra mutató mutató

Abban az esetben, ha a `*` bal oldalán van a `const` kulcszsó, akkor **konstansra mutató mutatóról** beszélünk. Ekkor a mutatón keresztül nem tudjuk módosítani a mutatott értéket, viszont a mutatót magát át tudjuk állítani, hogy egy másik memóriacímre mutasson. Deklarálásakor nem kell kezdőértéket adnunk neki, ilyenkor a fordító automatikusan `nullptr` étéket fog adni neki.

```
int main()
{
  // a két változat ugyanazt a viselkedést produkálja, a fordító nem tesz
```

```
különbséget
  // aközött, hogy az int-et vagy a const kulcsszót írjuk ki előbb
  const int* i = nullptr;
  int const* ii = nullptr;
}
```

Konstansra mutató mutatóval rátudunk mutatni konstans, illetve nem konstans adattagra is. 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.

Konstans mutató

Konstans mutatóról abban az esetben beszélünk, amikor a `const` kulcsszó a `*` karakter jobb oldalán van. Ilyenkor tudjuk módosítani a mutatott értéket a mutatón keresztül, viszont nem tudjuk át állítani a mutatót egy másik memóriacímre.

Fontos: már deklaráláskor meg kell adnunk a memóriacímet, amire szeretnénk vele mutatni és ez a program futása során nem módosulhat. Magyarán mondva, ahova inicializáljuk csak oda fog mutatni.

```
int main()
{
  int i = 1;
  int* const cPtrI = &i;
}
```

Konstansra mutató konstans mutató

Fennállhat olyan helyzet is, amikor egy mutató lehet egy konstansra mutató konstans mutató is, amin keresztül nem lehet megváltoztatni a mutatott értéket és a mutatót sem lehet máshova átállítani.

```
int main()
{
  int i = 1;
  const int* const ccPtrI= &i;
}
```

Tömbök (*arrays*)

A tömb a C++ egy beépített adatszerkezete, mellyel több **azonos típusú** elemet tárolhatunk és kezelhetünk egységesen.

Fontos: a tömbök sorfolytonosan helyezkednek el a memóriában.

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

Ebben az esetben, ha nem adjuk meg expliciten a `[]` operátorok között a tömb méretét a fordító ki fogja következtetni a példányosításból. Az `array` egy 5 elemű tömb. Nézzük meg mekkora a mérete bájtokban.

Figyelem: ez implementáció függő!

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

> kimenet: 20 4

Mivel az `array`-ben egész számokat tárolunk - egészen pontosan 5 darabot - láthatjuk, hogy az `array` mérete pontosan az ötszöröse az `int`-nek ezért mondhatjuk, hogy a tömbök tiszta adatok.

Próbáljuk meg kiíratni a tömb 6. elemét. Ehhez használjuk az **indexelő operátort** (`[]`). A tömb elemeinek elérésével bővebben a következő fejezet foglalkozik.

Megjegyzés: ez konstans idejű (`O(1)`) elem elérést tesz lehetővé.

```
int main()
{
  std::cout << array[6] << std::endl;
}
```

Szemmel látható, hogy túl fogunk indexelni, ez pedig nem definiált viselkedéshez vezet. Várhatóan memóriaszemetet fogunk kiolvasni az utolsó elem helyett, de ezt nem tudhatjuk pontosan. Az újabb fordítók már figyelmeztetnek a fordítás alatt, hogy potenciálisan túl indexelünk a tömbön. Mivel több memóriához nyúlunk hozzá, mint amihez kellene, nagyobb az esély arra, hogy futási idejű hibába ütközzünk. A programunk ilyen esetekben **szegmentálási hibával** (*segmentation fault*) állhat le. Ezen hibák elkerülése érdekében használhatunk **sanitizereket**. Ezek létrehoznak ellenőrzéseket, amik azelőtt észrevesznek bizonyos nem definiált viselkedéseket, mielőtt azok megtörténnének.

Fontos: a tömbön való túlindexelés nem definált viselkedés.

Tekintsük az alábbi programot:

```
#include <iostream>

int main()
```

```
{
  int arr[4] = {1,2,3,4};
  arr[4] = 5;
}
```

```
$ clang++ main.cpp -fsanitize=address
```

A sanitizerek csak abban az esetben találnak meg egy hibát, ha a probléma előfordul, azaz futási időben, nem fordítási időben ellenőriz.

```
$ ./a.out
```

A sanitizer ebben az esetben hibaüzenetet fog visszaadni.

Tömbök méretének meghatározása

Mint azt korábban láthattuk a tömb tiszta adat. A `sizeof()` operátor segítségével megtudjuk határozni a méretét. Egy olyan tömbben amelyben *n* darab *T* típusú elemet tárolunk a tömb mérete *n * sizeof(T)*. Ezt már csak le kellene osztanunk *sizeof(T)*-vel, azaz a tömbben tárolt típus méretével, tehát a képlet: `sizeof(array) / sizeof(T)`. Azonban nem biztos, hogy tudjuk, hogy milyen elemek vannak az `array` tömbben, így kicsit generikusabban megfogalmazva az előző képletet a *sizeof(T)* helyett osztjuk le a tömb első elemének méretével, *sizeof(array[0])*. Ezt megtehetjük, mert tudjuk, hogy a tömb azonos típusú elemeket tartalmaz. Tehát a helyes képlet a következő:

```
sizeof(array) / sizeof(array[0]);
```

Tömb elemeinek elérése

Egy tömb adott elemére többféle módon is hivatkozhatunk:

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

Ahhoz, hogy megértsük a fentebb látható egyenlőséget tudni kell, hogy a tömbök nevei C++-ban konvertálódnak a tömb első elemére mutató mutatóra és, mivel - ahogy az fentebb említettük - a tömbök sorfolytonosan helyezkednek el a memóriában ezért tudunk a `+` operátor segítségével ugrálni az adattagjaikon.

Megjegyzés: a fentebb látható egyenlőséget nevezzük **pointer aritmetikának**.

Tekintsük az alábbi két dimenziós tömböt:

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

Az első `[]` jelek között nincs megadva méret, mert a fordító az inicializáció alapján meg tudja állapítani. A második dimenzió méretének megadása kötelező! A pointer aritmetika ekvivalencia mátrixok esetén is érvényes, azaz fennállnak az alábbi egyenlősségek:

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

Paraméter átadási mód, visszatérési érték

A C++ három különféle paraméter átadási módot különböztet meg:

- **érték szerint** (*pass by value*)
- **referencia szerint** (*pass by reference*)
- **mutató szerint** (*pass by pointer*)

Az értékek átadásának demonstrálásához a `swap()` függvényt fogjuk használni.

Érték szerinti paraméter átvétel

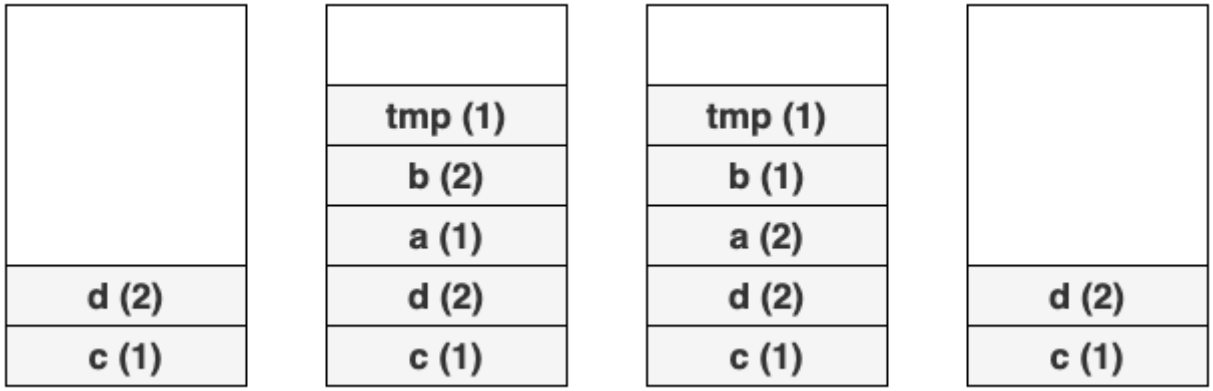
C++-ban ez az alapértelmezett paraméterátadási mód. Tekintsük a következő kódot:

```
#include <iostream>

void swapWrong(int a, int b)
{
  int tmp = a;
  a = b;
  b = tmp;
}

int main()
{
  int c = 1, d = 2;
  std::cout << c << ", " << d << std::endl;
  swapWrong(c, d);
  std::cout << c << ", " << d << std::endl;
}
```

> kimenet:
>
> 1 2
>
> 1 2

Megfigyelhető, hogy az első kiírás eredménye ugyanaz, mint a második kiírásé. Ez azonban egy teljesen jól definiált viselkedés. Ennek oka nem más mint, hogy **érték** szerint vettük át a paramétereket. Képzeljük el, hogy a stackbe a program berakja a `c` és `d` változókat. Ezután meghívja a `swapWrong()` függvényt, melyben létrehozott `a` és `b` paraméterek értékét megcseréli, de a függvényhívás után ezeket ki is törli a stackből. Az eredeti `c` és `d` változók értéke nem változott a függvényhívás során.

A stack tartalma érték szerinti paraméterátadás esetén.



Mutató szerinti paraméter átadás

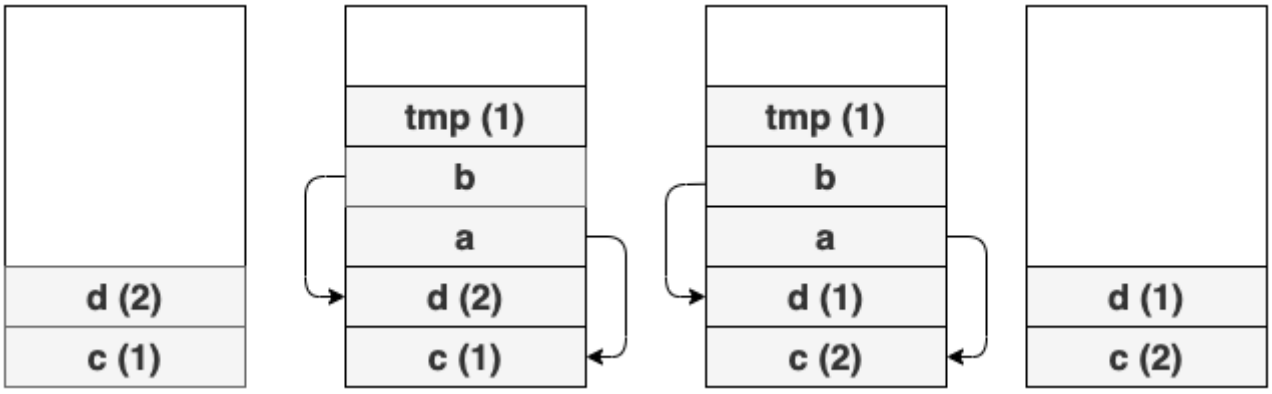
Írjuk meg helyesen a `swap()` függvényt.

```
#include <iostream>

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

int main()
{
  int c = 1, d = 2;
  std::cout << c << ", " << d << std::endl;
  swapPointer(&c, &d);
  std::cout << c << ", " << d << std::endl;
}
```

> kimenet:
>
> 1 2
>
> 2 1

Amennyiben ezt a függvényt hívjuk meg, valóban megcserélődik a két változó értéke. De ehhez fontos, hogy ne `swapPointer(c, d)`-t írjunk, az ugyanis fordítási idejű hibához vezetne, hiszen `c` és `d` változók típusa `int` és nem `int*`, márpedig mi mutatót adtunk meg bemenetként, amiben pedig referenciákat tudunk tárolni, így a helyes választás a `&c` és `&d` változók referenciája lesz.

A stack tartalma mutató szerinti paraméterátadás esetén.




```
#include <iostream>

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

int main()
{
  int c = 1, d = 2;
  std::cout << c << ", " << d << std::endl;
  swapPointerWRONG(&c, &d);
  std::cout << c << ", " << d << std::endl;
}
```

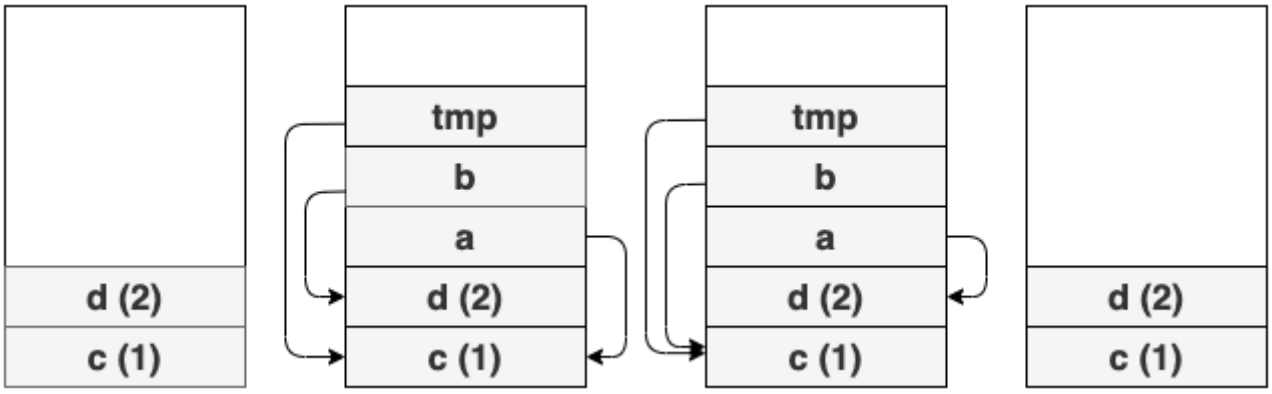
kimenet:

1 2

1 2

Ebben a példában nem a mutatók által mutatott értéket, hanem magukat a mutatókat cseréljük meg. Itt az fog történni, hogy a függvény belsejében a és b mutató másra fog mutatni. A mutatott értékek viszont nem változnak.

A stack tartalma rossz mutató állítása esetén.



Referencia szerinti paraméter átadás

Az előző példában láthattuk, hogy nem változtattuk meg, hogy mire mutassanak a mutatók, így azokat akár konstansként is definiálhattuk volna. Emlékezzünk vissza a konstans mutatók módosíthatják a mutatott értéket, de nem lehet őket átállítani egy másik memória címre.

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

Egy kis szintaktikai cukorkával megúszhatjuk azt, hogy folyton kiírjuk a `* const`-ot, mivel nem akarjuk megváltoztatni, hogy ilyen esetben a mutató hova mutasson. Erre való a referencia szerinti paraméter átadás. A referencia hasonlóan működik, mintha egy konstans mutató lenne, csak nem lehet sehova se mutató referenciát létrehozni, azaz nullreferenciát.

```
#include <iostream>

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

int main()
{
  int c = 1, d = 2;
  std::cout << c << ", " << d << std::endl;
  swapReference(c, d);
  std::cout << c << ", " << d << std::endl;
}
```

> kimenet:
>
> 1 2

> 2 1

Vegyük észre, hogy ilyenkor nem kell jelezni, hogy memóriacímet akarunk átadni, ezért `swapReference(c, d)`-t kell írnunk. Hátulütője a dolognak, hogy ha más valaki által írt függvényt akarunk használni, aminek nem ismerjük a szignatúráját nem tudjuk explicit megállapítani a hívásból, hogy az átadott paraméterek refenrecia vagy érték szerint lesznek átadva. C#-ban ennek a jelzésére van a `ref` kulcsszó.

Megjegyzés: bár referencia szerinti értékátadásnak nevezzük, de itt is történik másolás, a memóriacímet itt is érték szerint vesszük át.

Ökölszabály: ahol lehet ott használjunk referenciákat, ahol muszáj ott mutatót.

Paraméter átadási módok használata

érték szerinti: `f(int x)`

- a függvény nem módosíthatja a paramétert
- használd ezt, ha **könnyű másolni**
- Ökölszabály: ha a paraméter mérete legfeljebb 2- vagy 3 szó (word), azaz 32-bit, akkor érdemes érték szerint átadni
    - egyszerű primitív típusok esetén ajánlott, mint pl.: `int`, `double`, `char`, `bool`, stb.
    - Komplex típusok, saját osztályok, `std::string` és a különböző STL konténerek nem ajánlottak (továbbiakban ezeket a példákban T -vel jelöljük)

mutató szerinti: `f(T* x)`

- a függvény **módosíthatja a paramétert**,
- használd ezt, **ha költséges a másolás**,
- továbbá ha a **NULL lehet valid érték**

konstans mutató szerinti: `f(const * T x)`

- a függvény **nem módosíthatja a mutató által mutatott értéket**,
- használd ezt, **ha költséges a másolás**,
- továbbá ha a **NULL lehet valid érték**

referencia szerinti: `f(T& x)`

- a függvény **módosíthatja a paramétert**
- használd ezt, ha **költséges a másolás**,
- továbbá ha a **NULL NEM lehet valid érték**

konstans referencia szerinti: `f(const T& x)`

- függvény **nem módosíthatja a paramétert,**
- használd ezt, ha **költséges a másolás**
- továbbá ha a **NULL NEM lehet valid érték**
    - próbáljuk mindig ezt a változatot használni saját osztályokhoz, `std::string`-hez és STL adatszerkezetekhez, ha paramétert nem akarjuk módosítani
        - copy konstruktorok, copy assignmentek paraméterei

Megjegyzés: függvények bemeneti paramétereinek adhatunk alapértelmezett (*default*) értéket.

```
int f(int i = 0) { std::cout << i; }

int main()
{
  f();
  f(1);
}
```

> kimenet: 01

Ilyenkor, ha meghívjuk az `f()` függvényt és nem adunk neki bemeneti értéket, akkor az alapértelmezett értéket fogja használni. Ha nem adtunk volna az `f()`-nek alapértelmezett bemeneti paraméter értéket az első hívás szabálytalan lenne, mert nem találna a fordító olyan `f()` függvényt, melynek a szignatúrája megfelel annak a hívásnak.

Visszatérési érték problémája

Nem primitív típusoknál gyakran megeshet, hogy egy adott típushoz tartozó mutató 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.

```
#include <iostream>

std::string& concat(std::string& str_)
{
  str_ += " world";
  return str_;
}

int main()
{
  std::string str = "Hello ";
  std::cout << concat(str) << std::endl;
}
```

> kimenet: "Hello world"

Olyan típusoknál amelyeknél nagyon költésges a másolás (pl. `std::string`), célszerű referenciával átvenni és visszaadni a változókat. Primitív típusoknál nem érdemes referenciával átvenni adattagokat.

Tekintsük az alábbi programot:

```
#include <iostream>
```

```
std::string& concat(std::string& str_)
{
  std::string str = str_ + " world";
  return str;
}

int main()
{
  std::string str = "Hello ";
  std::cout << concat(str) << std::endl;
}
```

Mi ezzel a programmal a baj?

Ha jobban megnézzük a `concat()` függvény az `str` lokális változó referenciájával tér vissza. Ám, mint azt már tudjuk az `str` számára a stacken került foglalásra a memóriaterület, így függvény végén fel fog szabadulni, azaz törlődni fog a stackről. Így, amikor visszaadjuk a referenciáját, egy törölt változó referenciáját fogjuk visszaadni, ami valami memóriaszemét lesz. Ilyen esetekben a fordító egy figyelmeztetést ad: olyan objektumra hivatkozó referenciát adunk vissza, amely `concat()`-on belül lokális. Ez azt jelenti, hogy amint a vezérlés visszatér a `main()` függvényhez az `str` megsemmisül, és a `main()` függvény pedig az `str`-hez tartozó címen lévő értéket próbálná meg lemásolni. Mivel viszont az `str` 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.

Megjegyzés: mutatóknál ugyan ez a probléma áll fent.

Függvény átadása paraméterként

C++-ban lehetőségünk van arra is, hogy függvényeket adjunk át paraméterként. Ezeket az objektumokat **függvény mutatóknak** (*function pointer*) nevezzük. A függvénymutató egy függvény memóriacímét tároló változó, melyen a függvényhívás művelete (`()`) hajtható végre. Mikor lehetnek hasznosak? Nos, előfordulhat az a helyzet, hogy egy osztály ugyanazt a hívást sokféle objektum felé meg szeretné tenni, ám ezeknek **nincs közös őse**. Annak érdekében, hogy csökkentsük a függőségeket az osztályok között, nem hozunk létre egy közös őst, hanem függvénymutatókat használunk az osztályban.

```
#include <iostream>

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

void helper(int a, int b, int(*fptr)(int, int))
{
  std::cout << fptr(a,b) << std::endl;
```

```
}

int main()
{
  helper(1, 5, &multiply);
}
```

> kimenet: 5

Itt a `helper()` utolsó paraméterként egy olyan paramétert vár, amely igazából egy függvény, amely `int`-et ad vissza, és bemeneti paraméterül két `int`-et vár. Az `fptr` egy olyan függvényre mutató mutató, melynek két `int` paramétere van és a visszatérési értek is `int`.

Egy függvény átadásakor csak függvénypointert tudunk átadni. Egy függvénypointeren a függvény hívása az egyetlen értelemes művelet. Így az `&` jel elhagyható a függvény hívásakor és az `fptr` elől is elhagyható a `*` karakter a paramétereknél.

```
// ...
void helper(int a, int b, int fptr(int, int))
{
  // ...
}

int main()
{
  helper(1, 5, multiply);
}
```

> kimenet: 5

Függvény mutatókat értékül adhatunk változóknak is.

```
int multiply (int a, int b)
{
  return a * b;
}

int main()
{
  int (*mult)(int, int) = multiply;
  mult(1,2);
}
```

Megjegyzés: C++11 óta a `int (*mult)(int, int)` helyett használható az `auto` kulcsszó, amely fordítási időben végez típus kikövetkeztetést (*type deduction*), annak alapján mivel példányosítottuk az adott változót.

Az `auto`-tól egy régebbi megoldás a `typedef` használata.

```
// ...
int main()
{
  typedef int(*Multiply)(int, int);

  Multiply mult = multiply;
  mult(1,2);
}
```

A `typedef` egy speciális tárolási osztály, amely lehetővé teszi, hogy már létező típusokhoz újabb szinonim neveket rendeljünk.

A `typedef` helyett egy modernebb nyelvi eszköz C++11 óta a `using` kulcsszó. Az előző programot ennek a segítségével a következőképpen lehet leírni:

```
// ...
int main()
{
  using Multiply = int(*)(int, int);

  Multiply mult = multiply;
  mult(1,2);
}
```

Olvashatóbb kódot eredményez és template-ekkel is jobban használható.

Tömb, mint függvény paraméter

Próbáljunk meg egy tömböt érték szerint átadni egy függvénynek.

```
#include <iostream>

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

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

> kimenet: 20 8 (implementáció függő)

Amikor érték szerint próbálunk meg átadni egy tömböt, az átkonvertálódik a tömb első elemére mutató mutatóra. Emlékezzünk a pointer aritmetikánál tanultakra, azaz ebben az esetben az `f` függvényben található `sizeof(arr)` egy `int*` méretét fogja kiírni.

Hiába is próbálnánk megadni méretet a függvény bemeneti paraméterében

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

a kimenet akkor is egy mutató méretét fogja tartalmazni. Annak oka, hogy ez így működik visszanyúlik a C-vel való kompatibilitáshoz.

Megjegyzés: a szabvány szerint tömböt értékül adni nem is szabad.

Korábban már megismerkedtünk egy módszerrel, amely segítségével megtudjuk állapítani egy tömb méretét, elemszámát.

```
#include <iostream>

void f(int arr[], size_t arrSize) // arrSize-ban tároljuk a tömb méretét
{
  std::cout << sizeof(arr) << std::endl;
}

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

Megjegyzés: Amennyiben C++11-ben programozunk, érdemes az `std::array`-t használni, ami rendelkezik a tömb tulajdonságaival, viszont nem tud pointerré konvertálódni és mindig tudja a méretét.

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

Ha tömbre mutató mutatót szeretnénk, akkor a következőt írjuk:

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

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

méretére vagyunk kíváncsiak. Ha eltérő méretű tömböt próbálunk meg átadni, akkor nem fordul le a kód, mert nem tud egy 5 elemű tömbre mutató mutatóra konvertálódni például egy 6 elemű tömb.

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 mutatóra, és átadjuk külön paraméterben a tömb méretét, ahogy azt fentebb tettük.

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 mutatók ekvivalensek a tömbökkel. A tömb típusa tartalmazza azt az információt, hogy hány elemű a tömb. Egy mutató 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 mutatóra.

Statikus változók / függvények

Statikus változókat, függvényeket a `static` kulcsszó segítségével hozhatunk létre. A `static` kulcsszónak számos jelentése van, annak függvényében, hogy milyen kontextusban írjuk egy változó vagy függvény elé. Statikus változók a statikus globális tárterületen jönnek létre. Élettartamuk attól függ, hogy hol deklaráljuk őket és a program futásának végéig tart. Nem szükséges kezdőértékkel el látni őket csak abban az esetben, ha konstansok.

```
int main()
{
  static int i;       // ok
  static const int j; // nem ok, konstansoknak kezdőértéket kell adnunk
                      // emlékezzünk a konstans korrektségre
}
```

Fordítási egységre lokális változók

A függvényeken és osztályokon kívül deklarált statikus változók az adott fordítási egységre lokálisak – élettartamuk a futás elejétől futás végéig tart, és kizárólagosan az adott fordítási egységben láthatóak.

```
// main.cpp
static int x;

int main() { x = 1; }

// other.cpp
static int x;

void foo() { x = 2; }
```

Ha ezt a két fájlt együtt fordítjuk, nem kapunk linkelési hibát, ugyanis a `main.cpp`-ben lévő `x` egy teljesen más változó, mint ami az `other.cpp`-ben van.

Csak úgy, mint a globális változókra, fordítási egységen belül bármikor hivatkozhatunk egy statikus változóra, és hasonló módon inicializálódnak.

```
#include <iostream>

static int x;

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

> kimenet: 0

Függvényen belüli statikus változók

Azokat a változókat, melyek függvényen belül vannak a `static` kulcsszóval definiálva, **függvényszintű változónak** is szokás hívni. Élettartamuk a függvény első hívásától a program futásának végéig tart, míg láthatóságuk csak az adott függvényen belül van. A hagyományos lokális változókkal ellenben nem semmisülnek meg, amikor az adott függvény futása befejeződik. A következő kódrészlet szemlélteti ezt a viselkedést.

```
#include <iostream>

int f() {
  static int x = 0;
  return ++x;
}

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

> kimenet: 1 2 3 4 5

Ahogy az megfigyelhető, `x` csak egyszer inicializálódik, majd a későbbi függvényhívások után egyre növekszik az értéke.

Fordítási egységre lokális függvények

Nem csak változókat, függvényeket is deklarálhatunk statikusnak, melyek a fordítási egységre lokálisak.

```cpp
#include <iostream>

static int f() { return 0; }

int main()
{
  std::cout << f();
}
```

> kimenet: 0

Ezek a függvények csak az adott fordítási egységen belül érhetőek el.

Függvény túlterhelés (*function overloading*)

A C++ lehetőséget ad a **függvények túlterhelésére** (*overloading*), vagyis azonos nevű, de eltérő szignatúrájú függvények definiálására. Az ilyen függvénycsoportok hivásakor, a fordító a hívás szignatúrájából határozza meg, hogy melyik függvény hívódjon meg.

A C++ az alábbi attribútumokat érti bele egy függvény szignatúrájába:

- függvény neve
- osztály neve (tagfüggvényeknél)
- template argumentumok típusa (template függvényeknél)
- argumentumok típusa
- `const` típusmódosító (tagfüggvényeknél, tagfüggvényeket konstanság alapján is túl lehet terhelni).

Vegyük észre, hogy a szignatúrába a visszatérési érték típusa nem tartozik bele, vagyis ha két függvénydeklaráció pusztán visszatérési értékében különbözik, az nem számít túlterhelésnek, csak felüldefiniálásnak, és mint olyan legtöbbször fordítási hibát generál.

Tekintsük újra a `swap()` függvényt.

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

Ez a függvény egészen addig jól működik, amíg csak `int`-eket használunk. Mi a helyzet, ha két `std::string`-et szeretnénk megcseréli? A megoldás egyszerű, terheljük túl a `swap()` függvényt.

```cpp
void swap(std::string& a, std::string& b)
{
  std::string tmp = a;
  a = b;
```

```
    b = tmp;
}
```

Operátor túlterhelés (operator overloading)

A közönséges függvényekhez hasonlóan a legtöbb operátort is túl lehet terhelni, amely a felhasználói típusok kényelmesebb használatát teszi lehetővé (jellemző például az `<<` operátor túlterhelése).

Programozás során gyakran használunk operátorokat a programban elvégzendő feladatok, tömör, olvasható kifejezésére.

Az operátorok beépített típusok kezelésére kitűnően használhatóak, a programozó által létrehozott típusok, osztályok kezelésére azonban természetesen nem alkalmasak változatlan formában. A szabványos C++ operátorokhoz előre definiált úgynevezett operátor-függvények tartoznak. Amikor tehát C++-ban definiált műveleti jel értelmezését ki szeretnénk terjeszteni egy saját adattípusra, akkor a kérdéses operátor operátor-függvényére adunk meg egy újabb paraméter szignatúrájú változatot.

Túlterheléssel a programozó meghatározhatja, hogy mi történjék az általa létrehozott típusokkal az egyes operátorok hatására.

Fontos: túlterhelés során megváltoztathatjuk az egyes operátorok jelentését, de:

- nem hozhatunk létre új operátorokat
- nem változtathatjuk meg az operátorok aritását
- nem változtathatjuk meg az operátorok precedenciáját

Túlterhelhető operátorok: `+`, `-`, `*`, `/`, `%`, `^`, `&`, `|`, `~`, `!`, `=`, `<`, `>`, `+=`, `-=`, `*=`, `/=`, `%=`, `^=`, `&=`, `|=`, `<<`, `>>`, `>>=`, `<<=`, `==`, `!=`, `<=`, `>=`, `&&`, `||`, `++`, `--`, `->*`, `,`, `->`, `[]`, `()`, `new`, `new[]`, `delete`, `delete[]`.

Az `=` (értékadó), `&` (címképző) és a `,` (kiválasztó) operátorok túlterhelés nélkül is érvényesek.

Nem túlterhelhető operátorok: `::`, `.`, `.*`, `? :`, `sizeof`, `typeid`

Ezeknek az operátoroknak a túlterhelése nemkívánatos mellékhatásokkal járna, ezért nem lehet őket túlterhelni.

Túlterhelés menete:

A túlterhelés során az operátort megvalósító utasításokat függvényként adjuk meg. A függvény nevében az `operator` kulcsszót maga az operátor követi. Az operátor argumentumai a függvény argumentumai és visszatérési értéke, amelyek beépített típusok, objektumok (kis méret esetén) vagy referenciák (nagy méret esetén) lehetnek.

Az operátorok túlterhelésére használt függvények argumentumai lehetnek objektumok, de ez nem szerencsés nagyméretű objektumok esetében. Mutatókat nem használhatunk argumentumként, mert a mutatókra alkalmazott operátorok nem terhelhetők túl. A referenciák argumentumként használva lehetővé teszik a nagyméretű objektumok kezelését anélkül, hogy lemásolnánk őket.

Írjunk egy komplex számokat ábrázoló osztályt, majd terheljük túl a `+` és a `<<` operátorokat.

```
#include<iostream>

class Complex
{
public:
  Complex(int r_ = 0, int i_ = 0) { _real = r_; _imag = i_; }

  Complex operator+(Complex const &obj_)
  {
    Complex res;
    res._real = _real + obj_._real;
    res._imag = _imag + obj_._imag;
    return res;
  }

  friend std::ostream& operator<<(std::ostream &output_, const Complex
&comp_)
  {
    output_ << "R: " << comp_._real << ", I: " << comp_._imag;
    return output_;
  }

private:
  int _real;
  int _imag;
};

int main()
{
  Complex c1(10, 5);
  Complex c2(2, 4);
  Complex c3 = c1 + c2;
  std::cout << c3 << std::endl;
}
```

> kimenet: R: 12, I: 9

A ``<<` operátort azért fontos, hogy **barát**-ként adjuk meg az osztályon belül, mert anélkül szeretnénk meghívni, hogy új objektumpéldányt hoznánk létre.

`friend` mechanizmus

A friend mehcanizmus lehetővé teszi, hogy az osztály `private` és `protected` tagjait nem saját tagfüggvényből is elérjük. A `friend` deklarációt az osztályon belül kell elhelyezni tetszőleges elérésű részben. "barát" lehet egy külső függvény, egy másik osztály adott tagfüggvénye, de akár egy egész osztály is (vagyis annak minden tagfüggvénye).

```
class MyClass
{
public:
```

```
  friend int getCounter(const MyClass&);
  // friend int getCounter() const; -> hiba, non-member function cannot
have a const qualifier.
private:
  int _counter;
};

int getCounter(const MyClass& mc_) { return mc_._counter; }

int main()
{
  MyClass mc;
  getCounter(mc)
}
```

Literálok

Karakterlánc literálok

Mi lesz a **"Hello"** literál típusa?

Egy konstans karakterekből álló *6* méretű tömb (`const char[6]`). Azért 6 elemű, mert a karakterlánc literál végén el van tárolva a végét jelző `\0` karaktert.

Megjegyzés: változó típusát a `typeid(variable).name()` segítségével tudjuk lekérdezni.

H	E	L	L	O	\0

```
int main()
{
  char* hello = "Hello";
  hello[0] = 'B';
}
```

A fenti kódban megsértettük a konstans korrektséget, mert egy nem konstansra mutató mutatóval mutatuk egy konstans karakterlánc literál első elemére. Ennek ellenére, a fenti kód lefordul. Ennek az az oka, hogy az eredeti C-ben nem volt `const` kulcsszó, a kompatibilitás miatt ezért C++-ban lehet konstans karakterlánc literál elemire nem konstansra mutató mutatóval mutatni.

Megjgyzés: ezt a fajta kompatibilitás miatt meghagyott viselkedés **kerülendő**. A kód lefordul ugyan, de kapunk rá warningot.

Ha módosítani próbáljuk a karakterlánc literál értékét, az *nem definiált viselkedéshez* vezet. Linux operációs rendeszer futtatáskor futási idejű hibát kapunk, egészen pontosan szegmentálási hibát. Ennek oka, hogy a konstansok értékei **readonly** memóriában vannak tárolva, aminek a módosítását nem engedi az operációs rendszer.

Szám literálok

Függően attól, hogy hogyan írunk egy szám literált C++-ban, más lehet a jelentése.

`5`	`int`
`5.`	`double`
`5.f`	`float`
`5e-4`	`double`, értéke 0.0005
`5e-4f`	`float`
`0xFF`	16-os számrendszerben ábrázolt `int`
`012`	8-as számrendszerben ábrázolt `int`
`5l`	`long int`
`5u`	`unsigned int`
`5ul`	`unsigned long int`

Létezik C++-ban `signed` kulcsszó, mely a `char` miatt lett bevezetve. A `char` is egész számokat tartalmaz, de az implementáció függő, hogy a `char singed` vagy `unsigned` értéket tartalmaz-e.

Természetesen leggyakrabban egész számoknál használják. Alapértelmezetten minden `int` egy `signed int`, amennyiben egy előjel nélküli egész számra van szükségünk, `unsigned int`-et vagy röviden `unsigned`-ot kell írnunk. Megállapíthatjuk tehát, hogy az `unsigned` típusok ugyanannyi számjegyet tudnak tárolni, ám értékben kétszer akkorát.

Míg a túlcsordulás `signed` típusoknál *nem definiált viselkedés*, addig `unsigned` típusoknál definiált. Ekkor ugyanis a maximális érték (mely implementációfüggő) utáni inkrementálás 0-ra állítja a változót. Értelemszerűen, ez a determinisztikus viselkedés futási idejű költséggel jár.

C++-ban tizedes vessző helyett pontot kell írni ha lebegőpontos számot szeretnénk definiálni. Ennek ellenére az alábbi kód mégis helyesen lefordul:

```
int pi = 3,14;
```

Miért van ez?

Az ok az, hogy a vessző operátor (`,`) egy szekvenciapont is, így a fordító az egyenlőség bal oldalát amikor kiértékeli, először rendre kiértékeli a `3` számliterált, majd eldobja és utána kiértékeli a `14`-et, és értékül adja `pi`-nek.

A lebegőpontos számok másik veszélye, az összehasonlítás. **Minden lebegőpontos szám tartalmazhat egy kis pontatlanságot**.

```
for(double d = 0; d != 1; d += 0.1)
{
  std::cout << i << " ";
}
```

Az elvárt kimenet az lenne, hogy:

> 0 0.1 0.2 ... 1.0

de legnagyobb valószínűséggel végtelen ciklusba futunk. Ez azért van, mert a 0.1 (várhatóan) tartalmaz egy kis pontatlanságot, és így hiába írja ki a programunk hogy `d` értéke 1, az várhatóan csak nagyon közel lesz hozzá.

Megjegyzés: Viszonylag kevés esetben éri meg `float`-ot használni `double` helyett. Ennek oka, hogy a odern CPU-k ugyanolyan hatékonysággak képesek dolgozni mind a kettővel, így érdemesebb a pontosabbat választani. (Ha a GPU-t programozzuk, az lehet egy kivétel.)

Megjegyzés: Érdemes mindig `int`-et használni, ha nincs különösebben jó okunk arra, hogy mást használjunk. Az `int`-el általában a leghatékonyabb a processzor.

Struktúrák mérete

Primitív típusok mérete

Mint azt már korábban láttuk a `sizeof(char)` mindig 1-et ad vissza. A karakter mérete mindig az egység. Minden más típusra a `sizeof()` függvény azt adja vissza, hogy paraméterül megadott objektum vagy típus mérete hányszorosa a `char`-nak.

Attól, hogy `sizeof(char)` == `1`, a `char` mérete bájtokban még mindig **implementáció függő**.

A char méretén túl minden másnak a mérete implementációfüggő, bár a szabvány kimond pár relációt (bővebben a C++ adattípusok fejezetben olvashatunk erről):

`sizeof(X)` == `sizeof(signed X)` == `sizeof(unsigned X)`

`sizeof(float)` ≤ `sizeof(double)` ≤ `sizeof(long double)`

`sizeof(short)` ≤ `sizeof(int)` ≤ `sizeof(long)`

`sizeof(char)` ≤ `sizeof(bool)`

Nem primitív típusok mérete

Egy általunk megalkotott típus mérete több szabálytól is függhet.

```
#include <isotream>

struct Student
{
  double weight;
  int age;
  int height;
};

int main()
{
```

```
  std::cout << sizeof(double) << std::endl;
  std::cout << sizeof(int) << std::endl;
  std::cout << sizeof(Student) << std::endl;
}
```

A `double` mérete 8 bájt, az `int` mérete 4 bájt, `Student`-é 16, tehát a fenti kód lefuttatása után láthatjuk, hogy a `Student` struktúra tiszta adat.

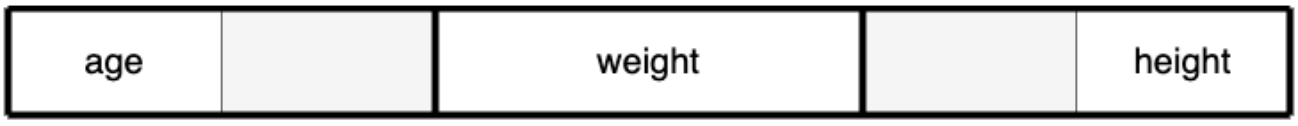
Rendezzük egy kicsit át mezők sorrendjét a következő módon

```
struct Student
{
  int age;
  double weight;
  int height;
};
```

A kód újbóli lefutása érdekes eredményt adhat. Ebben az esetben a `Student` már 24 bájtot foglal. Ennek az oka az, hogy míg az első esetben így volt eltárolva a memóriában: (ne feledjük, ez még mindig implementációfüggő !)

A `weight`, illetve az `age` és a `height` pont efértek 1-1 gépi szóban. Viszont, ha megcseréljük a sorrendet, ez már nem lesz igaz:

Itt a `weight` két különböző gépi szóba kerülne. Ez a ma használt processzorok számára nem hatékony, hiszen a `weight` értékének kiolvasásához vagy módosításához két gépi szót is olvasni vagy módosítani kéne (a legtöbb processzor csak szóhatárról tud hatékonyan olvasni).



A fenti elrendezés hatékonyabb, bár 3 gépi szót használ. Ebben az esetben a fordító úgynevezett **padding**-et illeszt be a mező után. Ennek hatására hatékonyan olvasható és módosítható minden mező. Cserébe

több memóriát foglal a struktúra.

A szabvány kimondja, hogy egy `struct` mérete az adattagok méreteinek összegénél nagyobb vagy egyenlő.

Az, hogy egy gépi szó mekkora, **implementációfüggő**.

OOP - Objektum-Orientált Programozás (*Object Oriented Programming*)

Az objektum-orientált programozás az objektumokat és köztük fennálló kölcsönhatásokat használja alkalmazások és számítógépes programok tervezéséhez. Négy dolog szükséges ahhoz, hogy egy programozási nyelvről azt mondhassuk, hogy támogatja az OOP-t:

- Adatrejtés (*data hiding*)
- Egységbezárás (*encapsulation*)
- Öröklődés (*inheritance*)
- Polimorfizmus - többalakúság (*polymorphism*)

Alapelemek

Osztály (*class*)

Az osztály meghatározza egy objektum elvont jellemzőit, beleértve az objektum jellemvonásait (attribútumok, mezők, tulajdonságok) és az objektum viselkedését (amit az objektum meg tud tenni, metódusok, műveletek, funkciók).

Azt mondhatjuk, hogy az osztály egy tervrajz, amely leírja valaminek a természetét. Például, egy Autó osztálynak tartalmazni kell az autók közös jellemzőit (gyártó, motor, fékrendszer, maximális terhelés stb.), valamint a fékezés, a balra fordulás stb. képességeket (viselkedés).

Osztályok önmagukban biztosítják a modularitást és a strukturáltságot az objektum-orientált számítógépes programok számára. Az osztálynak értelmezhetőnek kell lennie a probléma területén jártas, nem programozó emberek számára is, vagyis az osztály jellemzőinek „beszédesnek" kell lenniük. Az osztály kódjának viszonylag önállónak kell lennie (egységbezárás – encapsulation). Az osztály beépített tulajdonságait és metódusait egyaránt az osztály tagjainak nevezzük (C++-ban adattag, tagfüggvény).

Objektum (*object*)

Az osztály az objektum mintája. Az Autó osztály segítségével minden lehetséges autó típust megadhatunk, a tulajdonságok és a viselkedési formák felsorolásával.

Példány (*instance*)

Az objektum szinonimájaként az osztály egy adott példányáról is szokás beszélni. A példány alatt a futásidőben létrejövő aktuális objektumot értjük. Így elmondhatjuk, hogy az énAutóm az Autó osztály egy példánya. Az aktuális objektum tulajdonságértékeinek halmazát az objektum állapotának (*state*) nevezzük. Ezáltal minden objektumot az osztályban definiált állapot és viselkedés jellemez.

Metódus (*method*)

A metódusok felelősek az objektumok képességeiért. Mivel az énAutóm egy Autó, rendelkezik a fékezés képességével, így a Fékez() ez énAutóm metódusainak egyike. Természetesen további metódusai is lehetnek. A programon belül egy metódus használata általában csak egy adott objektumra van hatással. Bár minden autó tud fékezni, a Fékez() metódus hívásával csak egy adott járművet szeretnénk lassítani. C++ nyelven a metódus szó helyett a tagfüggvény kifejezést használjuk.

Alapvető elvek

Adatrejtés, egységbe zárás - data hiding, encapsulation

A fentiekben láttuk, hogy az osztályok alapvetően állapotokból és metódusokból épülnek fel. Azonban az objektumok állapotát és viselkedését két csoportba osztjuk. Lehetnek olyan jellemzők és metódusok, melyeket elfedünk más objektumok elől, mintegy belső, privát (*private*) vagy védett (*protected*) állapotot és viselkedést létrehozva. Másokat azonban nyilvánossá (*public*) teszünk. Az OOP alapelveinek megfelelően az állapotjellemzőket privát eléréssel kell megadnunk, míg a metódusok többsége nyilvános lehet. Szükség esetén a privát jellemzők ellenőrzött elérésére nyilvános metódusokat készíthetünk.

Általában is elmondhatjuk, hogy egy objektum belső világának ismeretére nincs szüksége annak az objektumnak, amelyik használja. Például, az Autó rendelkezik a Fékez() metódussal, amely pontosan definiálja, miként megy végbe a fékezés. Az énAutóm vezetőjének azonban nem kell ismernie, hogyan is fékez a kocsi.

Minden objektum egy jól meghatározott interfészt biztosít a külvilág számára, amely megadja, hogy kívülről mi érhető el az objektumból. Az interfész rögzítésével az objektumot használó, ügyfél alkalmazások számára semmilyen problémát sem jelent az osztály belső világának jövőbeli megváltoztatása.

Öröklődés - inheritance

Öröklés során egy osztály specializált változatait hozzuk létre, amelyek öröklik a szülőosztály (alaposztály) jellemzőit és viselkedését, majd pedig sajátként használják azokat. Az így keletkező osztályokat szokás alosztályoknak (*subclass*), vagy származtatott (*derived*) osztályoknak hívni.

Például, az Autó osztályból származtathatjuk a Ferrari és a Mercedes alosztályokat. Az énAutóm ezentúl legyen a Mercedes osztály példánya! Tegyük fel továbbá, hogy az Autó osztály definiálja a Fékez() metódust és az fékrendszer tulajdonságot! Minden ebből származtatott osztály örökli ezeket a tagokat, így a programozónak csak egyszer kell megírnia a hozzájuk tartozó kódot. Az alosztályok meg is változtathatják az öröklött tulajdonságokat. A származtatott osztályokat új tagokkal is bővíthetjük.

Az öröklés valójában „egy” (**is-a**) kapcsolat.

A többszörös öröklés (*multiple inheritance*) folyamán a származtatott osztály, több közvetlen ősosztály tagjait örökli. Például, egymástól teljesen független osztályokat definiálhatunk Autó és Hajó néven. Ezekből pedig örökléssel létrehozhatunk egy Kétéltű osztályt, amely egyaránt rendelkezik a teherautók és hajók jellemzőivel és viselkedésével. A legtöbb programozási nyelv (Java, C#) csak az egyszeres öröklést támogatja, azonban a C++-ban mindkét módszer alkalmazható.

Polimorfizmus - polymorphism

A polimorfizmus lehetővé teszi, hogy az öröklés során bizonyos (elavult) viselkedési formákat (metódusokat) a származtatott osztályban új tartalommal valósítsunk meg, és az új, lecserélt metódusokat a szülő osztály tagjaiként kezeljük.

Példaként tegyük fel, hogy az Autó és a Kerékpár osztályok öröklik a Jármű osztály Gyorsít() metódusát. A Teherautó esetén a Gyorsít() parancs a GáztAd() műveletet jelenti, míg Kerékpár esetén a Pedáloz() metódus hívását. Ahhoz, hogy a gyorsítás helyesen működjön, a származtatott osztályok Gyorsít() metódusával felül kell bírálnunk (*override*) a Jármű osztálytól örökölt Gyorsít() metódust. Ez a felülbíráló polimorfizmus.

A legtöbb OOP nyelv a parametrikus polimorfizmust is támogatja, ahol a metódusokat típusoktól független módon, mintegy mintaként készítjük el a fordító számára. A C++ nyelven sablonok (templates) készítésével alkalmazhatjuk ezt a lehetőséget.

Struktúrák (`struct`)

A C++ nyelven a struktúra (`struct`) típus több tetszőleges típusú (kivéve `void` és a függvénytípus) adatelem együttese. Ezek az adatelemek, melyek szokásos elnevezése **struktúraelem** vagy **adattag** (*data member*), csak a struktúrán belül érvényes nevekkel rendelkeznek. (A más nyelveken a mező (field) elnevezést alkalmazzák, mely elnevezést azonban a bitstruktúrákhoz kapcsolja a C++ nyelv.) Struktúrák záró kapcsos zárójele után mindig kell pontosvesszőt rakni (`;`).

Alapértelmezett módon a struktúrák adattagjai és metódusai publikusak a külvilág számára és a pont operátor (`.`) segítségével hivatkozhatunk rájuk a struktúrából létrehozott objektumpéldány nevén keresztül.

C stílusú struktúrák

```
typedef struct
{
  const char* name;
  int age;
  double salary;
} Person;

int main()
{
  Person person; // objektumpéldány létrehozása

  // adattagok inicializálása
  person.name = "Jack";
  person.age = 26;
  person.salary = 1500.50;
}
```

C++ stílusú struktúrák

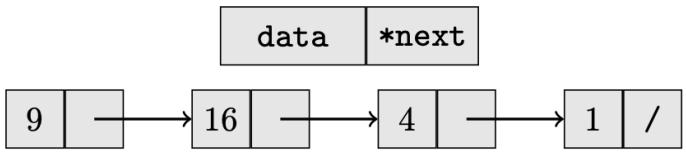
```cpp
struct Person
{
  std::string name;
  int age;
  double salary;
};

int main()
{
  Person person; // objektumpéldány létrehozása

  // adattagok inicializálása
  person.name = "Jack";
  person.age = 26;
  person.salary = 1500.50;
}
```

Mivel a struktúráknak tetszőleges adattagjaik lehetnek, így lehetőségünk van több struktúra egymásba ágyazására.

Feladat: láncolt listák

A következőkben egy láncolt listát fogunk implementálni, amely jól mutatja majd a dinamikus memóriakezelés veszélyeit.

A láncolt lista nevéből eredendően nem tömbszerűen (egymás melletti memóriacímeken) tárolja az objektumokat, hanem egymástól független memóriacímeken. Ezt úgy oldja meg, hogy minden adathoz rendel egy mutatót is, mellyel a következő listaelemet lehet elérni. A lista utolsó elemében a mutató a rákövetkező elem memóriacíme helyett `nullpointer` értéket vesz fel.




```cpp
struct List
{
  int data;
  List* next;
};

int main()
{
  List* head = new List;
  head->data = 9; // (*head.)data == head->data
  head->next = new List;

  head->next->data = 16;
```

```
  head->next->next = new List;
  head->next->next->data = 4;
  head->next->next->next = NULL; // modern C++-ban nullptr

  delete head;
  delete head->next;
  delete head->next->next;
}
```

Mondhatnánk, hogy készen is vagyunk, implementáltuk a láncolt listát - noha a használata nem túl kényelmes. Sajnos azonban ez a program hibás. A törlés sorrendje rossz: először töröljük a fejelemet (mely az első elemre mutat), viszont az első elem segítségével tudnánk a többi elemet elérni, így mikor a második listaelemet törölnénk, `head` már egy felszabadított memóriaterületre mutat. Ezt törlés utáni használatnak (*use after delete* vagy *use after free*) szokás nevezni és nem definiált viselkedés. A helyes megoldás:

```
int main()
{
  // ...
  delete head->next->next;
  delete head->next;
  delete head;
}
```

Fontos: a heap-en arra is figyelnünk kell, hogy jó sorrendben szabadítsuk fel a memóriát. Ha rossz sorrendben szabadítjuk fel az objektumokat, könnyen a fentihez hasonló hibát vagy memória szivárgást okozhatunk.

Ez a "láncolt lista” eddig elég szegényesen néz ki. A fő gond az, hogy nagyon sokat kell írni a használatához. Ezzel megszegjük a **DRY** programozás elvét: **Don't Repeat Yourself**. Itt sokszor írjuk le közel ugyanazt a kifejezést. Erre találnunk kell egy egyszerűbb megoldást. Írjunk egy függvényt, melynek segítségével új listaelemet hozhatunk létre.

```
List* add(List* head, int data)
{
  if (head == 0)
  {
    List *ret = new List;
    ret->data = data;
    ret->next = 0;
    return ret;
  }

  head->next = add(head->next, data);
  return head;
}
```

Ez egy olyan rekurzív függvény, mely addig hívja saját magát, míg a paraméterként kapott lista végére nem ér (azaz a head egy nullpointer). Amikor oda elér, létrehoz egy új listaelemet és azt visszaadja. A rekurzió felszálló ágában a lista megfelelő elemeit összekapcsolja.

Írjunk egy újabb függvényt, amely segítségével lehetőségünk lesz felszabadítani a lista által birtokolt memóriát.

```
void free(List* head)
{
  if (head == 0)
  {
    return;
  }

  free(head->next);
  delete head;
}
```

Itt a rekurzió szintén a lista végéig megy. A rekurzió felszálló ágában történik a listaelemek felszabadítása. Ennek az oka, hogy a felszabadítás a megfelelő sorrendben történjen meg.

Megjegyzés: a rekurzív függvények nem olyan hatékonyak, mint az iteratív (pl. `for` vagy `while` ciklus) társaik. Továbbá a sok függvényhívás könnyen stack overflow-hoz vezetnek. Egy rekurzív függvényt mindig át lehet írni iteratívvá.

Beszéljünk arról, mennyi a teher a felhasználón. Eddig tudnia kellett, milyen sorrendben kell felszabadítani az elemeket a listán belül, de most már elég arra figyelnie, hogy a lista használata után meghívja a `free()` függvényt. A felhasználó így kisebb eséllyel követ el hibát. **Legyenek a függvényeink és osztályaink olyanok, hogy könnyű legyen őket jól használni, és nehéz legyen rosszul.**