Informatika 3
2. előadás

Tóth Dávid

Budapesti Műszaki és Gazdaságtudományi Egyetem

2026.02.24.

Konverzió

- Előfordul, hogy a változó típusa, amiben az adatot tároljuk, nem megfelelő egy feladat elvégzéséhez.
- Pl. ha `a` és `b` `int` típusú változók, akkor az `a/b` művelet a tört egészrészét adja vissza.
- Ha az `a/b` tört értékére vagyunk kíváncsiak (azt pl. egy lebegőpontos változóban szeretnénk eltárolni), akkor típuskonverziót kell végeznünk: ha az osztásban legalább az egyik változó lebegőpontos, akkor a végeredmény is az lesz.
- A típuskonverziót a `static_cast<új típus>(változó)` operátorral végezhetjük el.
- Pl. az `a` `int` típusú válotzót a `static_cast<double>(a)` kód `double` típusúvá konvertálja.
- A `C`-ben a `double(változó)` kód teszi ugyanezt.

Konverzió

- 1. alapelv: kerüljük a típuskonverziót, amennyiben lehetséges
- ok: hibaforrás lehet, nagyon sokszor rosszul megtervezett program jele
- 2. alapelv: használjunk `static_cast`-ot a `C` stílusú konverzió helyett
- ok: könnyebben olvasható kód, biztonságosabb, a fordító ellenőrzi, sok hibát már fordítási időben el lehet csípni, a `C` stílusú konverzió hatása nem mindig meghatározott, a hiba csak futási időben derül ki
- (ezek később jelenthetnek majd gondot, de jobb az elején megszokni a jó gyakorlatot)

Legnagyobb közös osztó

- Probléma: hány olyan számpár van az $\{1, 2, \ldots, n\}$ halmazban, amelyek relatív prímek egymáshoz?
- Legyen p_n ezek aránya az összes pár közt, azaz a relatív prímség valószínűsége egy véletlen választás esetén.
- Létezik $\lim_{n\to\infty} p_n$? Ha igen, mihez konvergál?
- Határozzuk meg konkrét példákon!
- Írjunk egy gcd függvényt, majd számoljuk ki minden párra a fenti halmazban, és határozzuk meg a relatív prímek arányát!

GCD függvény

- hozzunk létre egy saját fejállományt: gcd.h
- itt fog szerepelni a függvény deklarációja
- a definíciót egy külön (ugyanolyan nevű) forrásfile-ba (cpp) írjuk
- a deklarációkat a fejállományban (`gcd.h`) a következő kód közé írjuk:

```
#ifndef GCD_H
#define GCD_H
...
#endif
```

- ez biztosítja, hogy a file tartalma ne kerüljön be többször a programba: ha a `GCD_H` szimbólum már definiálva volt, az `#infndef` és `#endif` közti rész nem másolódik be, ha még nem volt definiálva, definiáljuk

Névterek

- A C++ sztenderd könyvtára is kínál egy `gcd` függvényt, mely a `<numeric>` fejállományban található (C++17-től).
- Ahhoz, hogy azonos nevű és azonos számú és típusú paraméterekkel rendelkező függvényeket egyértelműen elérjünk, ún. *névtereket* (`namespace`) használunk.
- A névterek megadása:

```
namespace név {
    ...
}
```

- A kapcsos zárójelek közt deklarált függvények/objektumok az adott névtérbe kerülnek.
- A névterekben lévő függvények elérése:
  `névtérnév::függvény(paraméterek);`

Névterek

- A sztenderd könyvtár függvényei (és objektumai) az `std` névtérben vannak, elérésük:
  `std::függvénynév(paraméterek);`
- Ahhoz, hogy a névtér kiírása nélkül is elérjük egy adott névtér függvényeit, írjuk a következő sort a kódba:

  ```
  using namespace név;
  ```

- A FENTI MEGOLDÁS ROSSZ GYAKORLAT!
  RÖVID (PÉLDA)PROGRAMOKTÓL ELTEKINTVE
  NE HASZNÁLJUK!
- Ha ugyanis ezt alkalmazzuk, és több névtérben van deklarálva egy függvény azonos névvel, akkor esetenként nehezen követhető nyomon, hogy melyik hívódik meg.

Tömbök

- Egy lista tárolásának legegyszerűbb módja (C-ben vagy) C++-ban a tömbök használata.
- Deklarálás: `típus név[elemszám];`
- Tömbök elemeire így hivatkozunk: `név[index]`.
- Az indexelés 0-tól indul, n elemű tömb utolsó elemének indexe $n-1$.
- Értékadás:

```
int tomb[3];
tomb[0] = 1;
tomb[1] = 2;
tomb[2] = 3;
```

- Inicializálni így is lehet: `int tomb[] = {1, 2, 3};`
- Inicializálható az összes elem 0 értékkel (C++11-től):

```
int tomb[100] = { 0 };
```

Konstansok

- Egy tömb elemszámát meg kell adni a deklarálásnál.
- Az elemszám nem lehet egy változó értéke, kivéve, ha az egy fordítási időben ismert értékű *konstans*.
- Konstans változókat a `const` kulcsszó segítségével definiálhatunk, pl.:

```
const int tomb_elemszam = 3;
int tomb[tomb_elemszam];
```

- Ez a kód működik, mert a `tomb_elemszam` változó értéke a definiálás után már **nem változhat**, ezt az értéket a fordító már fordításkor ismeri.

Konstansok

- Tipp: messziről kerüljük a "mágikus konstansokat", azaz az olyan számokat a kódban, amiknek a jelentése az olvasó számára nem világos.
- Ehelyett definiáljunk konstans változókat, amiknek a neve elárulja a szám jelentését, és a szám helyett mindig használjuk ezeket.
- További előny: ha a szám változik, csak egy helyen kell átírni!

```
const int tomb_elemszam = 3;
int tomb[tomb_elemszam];
for(int i = 0; i < tomb_elemszam; ++i) {
    tomb[i] = i + 1;
}
```

Karakterlánc-literálok

- Karakterláncokat tárolhatunk `char` tömbben:

```
char lanc[] = "Maci Laci";
```

- A tömb hosszát megkaphatjuk a `sizeof(név)` paranccsal.
- Futtassuk a következő kódot! Mit tapasztalunk?

```
char lanc[] = "Maci Laci";
for(int i = 0; i < sizeof(lanc); ++i) {
    cout << "A(z) " << i+1 << ". karakter: "
        << lanc[i] << endl;
}
```

- Az idézőjelek közé írt karakterlánc `const char` típusú elemekből álló tömb, amely a végén tartalmaz egy extra '\0' karaktert, ez jelzi a karakterlánc végét. Az előző példában a `lanc` tömb hossza 10.
- Az ilyen karaterláncokat *C stílusú karakterláncoknak*, röviden *C stringeknek* nevezzük.

Karakterlánc-literálok

- A következő példában az első tömb nem C string, a második viszont igen:

```
char str[]  = {'M', 'a', 'c', 'i', ' ' };
char cstr[] = {'L', 'a', 'c', 'i', '\0'};
```

- Próbáljuk meg a következő C++ kóddal kiíratni a két tömböt. Mit tapasztalunk?

```
cout << str << endl;
cout << cstr << endl;
```

- Próbáljuk ki, hogy mi történik, ha egy egészekből álló tömböt íratunk így ki:

```
int egesz[] = {1, 2, 3};
cout << egesz << endl;
```

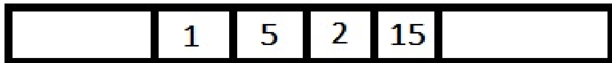
- Nem azt kaptuk, amit vártunk...pl. az utolsó sor kimenetele lehet: "000000F6F7AFF5F8" - ez micsoda?

Pointerek

- A tömbök neve C++-ban egy ún. *pointer/mutató*.
- A számítógép a változók értékeit a memóriában tárolja.
- A memóriában tárolt minden egyes byte-nak van egy *címe*, amelyet egy szám ír le.
- Minden pointer egy ilyen címet tárol, a típusa pedig elárulja, hogy milyen adatot tárolunk az adott helyen.
- Pl. a fenti egesz tömb neve egy olyan pointer, amely egy egész számot, nevezetesen a tömb első elemét tartalmazó memóriacímre mutat.

- A tömb elemei a memóriában szomszédos blokkokban tárolódnak:

```
int t[] = {1, 5, 2, 15};
```



- Itt `t` az 1-et tartalmazó blokk címére mutat, az innentől számolt 4 byte-on tárolódik az 1 érték (ha az `int` 4 byte-os).
- A `t[i]` jelölés arra a tömbelemre (és nem a címére!) hivatkozik, ami a `t` címtől `sizeof(int)*i` byte-ot tovább (az ábrán jobbra) lépve kapott memóriacímmel kezdve tárolunk.

Pointerek

- A tömbelemeket tehát egyesével tudjuk kezelni (változtatni, kiírni):

```
int t[] = {1, 5, 2, 15};
for(int i = 0; i < sizeof(t); ++i) {
    t[i] += 2;
    cout << t[i] << endl;
}
```

- A következő sor a `t` tömb első elemének címét írja ki:

```
cout << t << endl;
```

- Ha egy `char` tömb első elemének címét adjuk oda a `cout`-nak, akkor úgy veszi, hogy egy C stringet akarunk kiíratni, és az első '\0' karakterig sorban kiírja az adott címtől kezdve memóriában tárolt adatot.
- Egy C string hosszát az `strlen` függvénnyel határozhatjuk meg (`cstring` fejállomány): ez az első `\0` karakterig szereplő karakterek számát adja meg.

- Figyelem! A következő kód lefordul:

```
int t[] = {1, 5, 2, 15};
cout << t[4];
```

- A `t` tömbnek csak 4 eleme van, de a `t` címtől kezdve számolt ötödik 4 byte-os blokkban tárolt elemre is lehet hivatkozni, a program lefordul. A memória ezen részének a tartalma azonban nem ismert, a futás kimenetele nem meghatározott.

Pointerek

- Pointereket deklarálhatunk, módosíthatunk, hivatkozhatunk az általa mutatott változóra:

```
int t[] = {1, 5, 2, 15};
int* i = t;
*i = 3;
i = &t[1];
i = i + 2;
```

- Az `i` mutató egy int típusú változó címét tartalmazza.
- `*i` az `i` címen tárolt változóra vonatkozik, ennek értékét 3-ra változtatjuk.
- Az `i` mutatót a tömb második elemének címére állítjuk.
- Az `i` mutatót eltoljuk a memóriában előre `2*sizeof(int)` byte-tal, így a `t` tömb 4. elemére fog mutatni.

Pointerek

- A következő kód nem fordul:

```
const int a = 20;
int* i = &a;
```

- Az a változó konstans, a címét nem adhatjuk értékül egy egészre mutató pointernek, mert a pointeren keresztül változtatni lehetne rajta!
- A következő kód helyes:

```
const int a = 20;
const int* j = &a;
```

- A `j` pointer egy `const int` konstans változóra mutat, ilyenen keresztül nem lehet változtatni a változó értékét.

Pointerek

- Mit jelenthet a következő deklaráció?

```
const int a = 20;
const int *const j = &a;
```

- A fenti utolsó sorban az első `const int` azt jelenti, hogy a pointer egy konstans változó értékére mutat, ezt az értéket tehát nem lehet változtatni.
- A második `*const` azt jelenti, hogy a pointer maga konstans, tehát az általa tárolt címet nem lehet megváltoztatni.

Pointerek

- Ha értéket akarunk adni egy pointernek, de nem áll rendelkezésre memóriacím, akkor beállíthatjuk a null pointert értékként, ami egyszerűen azt jelenti, hogy nem mutat a pointer konkrét címre:

```
const int* p = nullptr;
```

- Ennek segítségével könnyen ellenőrizhető, hogy a pointer értelmes címre mutat-e:

```
int* p = nullptr;
...
if(p){
// ez a rész akkor fut le, ha p nem null pointer
}
```

Pointerek

- Pointerekkel dolgozni körültekintést igényel: könnyen futás idejű hibát eredményezhetnek, ezek nehezen detektálhatók.
- De: hatékony kódot lehet velük írni!

Rendezés

- Tömböket az `algorithm` fejállományban lévő `sort` függvénnyel rendezhetünk:

```
int tomb[] = { 1, 2, 10, -4, 6 };
sort(tomb, tomb + 5);
```

- A `sort` első paramétere egy pointer a rendezendő tömb elejére, a második paraméter egy pointer a tömb utolsó eleme *utáni* memóriacímre.
- Ez a kód növekvő sorrendbe rendezi a tömböt.
- Csökkenő sorrendbe rendezés:

```
sort(tomb, tomb + 5, greater<>());
```

Tömb visszaadása

- Egy függvény visszatérési értéke nem lehet tömb.
- Ehelyett paraméterként megadhatunk egy tömb elejére mutató pointert, ahová az output kerül.
- A következő függvény egy bemeneti tömb első *n* elemét írja fordított sorrendben a kimeneti tömb első *n* elemébe:

```
void fordit(const int* input, int n, int* output){
    for(int i = 0; i < n; ++i) {
        output[i] = input[n - 1 - i];
    }
}
```

- Az `input` tömb elejére mutató pointer konstans, ez azt jelzi, hogy a függvény azt nem változtatja.
- A felhasználónak kell gondoskodnia róla, hogy az `n` paraméter ne legyen nagyobb az `input` tömb méreténél, ill. hogy az `output` tömb elég nagy legyen.

Dinamikus memóriafoglalás

- Tömb deklarálásánál a méret nem lehet egy (nem konstans) változó értéke.
- De tudunk dinamikusan memóriát foglalni a `new` kulcsszóval:

```
int n = 5;
int* t;
t = new int[n];
```

- A fenti kód harmadik sora az `n` változó által tárolt értéknyi `int`-nek elegendő memóriablokkot foglal, a `t` pointer pedig ezen blokk elejére fog mutatni.
- Fontos: a `new`-val foglalt memóriát az adott függvényben fel *kell* szabadítanunk a `delete` kulcsszóval, tömb esetén:

```
delete[] t;
```

Dinamikus memóriafoglalás

- Nem csak tömböt lehet így foglalni:

```
int* p;
p = new int;
...
delete p;
```

- Ha nem használjuk a `delete` kulcsszót, akkor a `new`-val lefoglalt memória nem szabadul fel, ha túl sokszor lefut a kód, elfogy a foglalható memória (memóriaszivárgás).
- Már felszabadított memóriaterületre soha ne használjuk a `delete`-et!
- Összefoglalva: minden `new`-hoz tartozzon pontosan egy `delete` az adott függvényben.