Bevezetés a programozásba

8. Előadás

Függvények-2

Paraméterek érvényessége és láthatósága

- Szabad, és nem kötelező ugyanazokat a változóneveket használni aktuális és formális paraméterként
- A függvény paramétere lokális változó, ami csak a függvényben él
- Minden olyan változó, ami függvényben lett deklarálva, lokális változó
- A main() függvény változói sem használhatóak a többi függvényben
- A függvények paraméterekkel kommunikálnak

Paraméterek érvényessége és láthatósága

```
#include <iostream>
#include <cmath>
using namespace std;

double terulet(double a, double b, double c)
{
    double s = (a+b+c)/2.0;
    return sqrt((s-a)*(s-b)*(s-c)*s);
}

int main() {
    double ha, hb, hc;
    cin >> ha >> hb >> hc;
    double t = terulet(ha, hb, hc);
    cout << "Terulet: " << t <<endl;
    return 0;
}
```

Paraméterek érvényessége és láthatósága

```
#include <iostream>
#include <cmath>
using namespace std;

double terulet(double a, double b, double c)
{
    double s = (a+b+c)/2.0;
    return sqrt((s-a)*(s-b)*(s-c)*s);
}

int main() {
    double a, b, c;
    cin >>  a >>  b >>  c;
    double t = terulet(a,  b,  c);
    cout << "Terulet: " << t <<endl;
    return 0;
}
```

Névválasztás

- Érdemes a formális paraméter nevét a függvénybeli szerep szerint elnevezni
- Az aktuális paraméter lehet akár kifejezés is
- Néha előfordul, hogy ugyanolyan nevű változók vannak az aktuális paraméterben és a formális paraméterben
  - Ez nem baj, amíg tudjuk hogy ezek külön változók

Változó láthatósága

- Egy változó látható a programkód azon soraiban, ahol a nevének leírásával használható
  - C++ nyelvben a lokális változók nem láthatóak más függvényekben
  - A mindenhol látható változót függvényen kívül kell deklarálni: globális változó
    - A globális változókat ahol lehet, kerüljük. Néhány értelmes használata van, ezekre általában kitérünk majd, minden másnál a jó stratégia az, hogy paraméterben adjuk át a szükséges adatokat
    - Ha mégis használjuk, azokban a függvényekben lesz látható, amelyek a deklaráció után vannak

Változó érvényessége

- A változó érvényes a program futásának azon időintervallumában, amikor az értékét megtartja
- C++-ban a lokális változó a deklarálásától számítva addig él, amíg a deklarálásának blokkja be nem fejeződik
- A paraméterek a függvény végéig élnek

Kitérő a láthatóságról

- Nem lehet egy blokkban két egyforma nevű változót deklarálni, de lehet már látható változó nevét használni
- Shadowing: a frissen deklarált változó eltakarja a többi ugyanolyan nevű láthatóságát

```
#include <iostream>
using namespace std;

int a=1;
int main()
{
    int a=2;
    {
        int a=3;
        cout << a << endl;//3
    }
    cout << a <<endl; //2
    cout << ::a << endl; //1
    return 0;
}
```

Procedurális (vagy hierarchikus) programozás

- Programtervezési elv: a feladatot osszuk részfeladatokra és mondjuk függvényekben csoportosítsuk azokat
  - „dekompozíció”, „redukció”
- int nagyonbonyolultfeladat(P1, P2, P3, P4, …){
  egyikkicsitegyszerűbbfeladat(P2, P4, …);
  másikkicsitegyszerűbbfeladat(P1, P2, …);
  }
- Top-down vagy Bottom-up felépítés
- 80-as évekig nagy szoftvereknél egyeduralkodó

Trükk

- Mi történik, ha a függvényhívások „körbeérnek”?
  - Excel: „feloldhatatlan körbehivatkozás”
    - OpenOffice: „522 körkörös hivatkozás; A képlet követlenül vagy közvetetten önmagára hivatkozik, és az Eszközök - Beállítások - OpenOffice.org Calc - Számítás panel **Iterációk** beállítása nincs kijelölve.„
    - A C++ program így könnyen végtelen ciklusba kerülhet, de ezen lehet segíteni
- Rekurzió, rekurzív függvény
- Eleinte nehéz lehet megérteni, hogy egy függvény több példányban is várjon visszatérési értékre

Rekurzív függvény

```
int faktor( int p )
{
     if( p>1 ) {
          return p*faktor( p-1 );
     } else {
          return 1;
     }
}

int main()
{
     cout << faktor( 5 ) << endl;
     return 0;
}
```

Összefoglalás I.

- A nagyobb problémákat függvényekre bontjuk
- Az ismétlődő kódrészleteket egyszer írjuk csak le és az eredeti több helyről meghívjuk
- A változók a másik függvényben nem láthatóak, ezért paraméterekkel és visszatérési értékekkel kommunikálunk
- Érvényesség, láthatóság, lokális változók
- Függvények hívhatják egymást

Nyitott kérdések

- A sorok beolvasását a **getline(cin, s)** függvénnyel hajtuk végre. Ez miért nem úgy néz ki, hogy **s = getline(cin)**?
- Hogyan tud a függvény több eredményt visszaadni?
- Hogyan kell tömböt paraméterben átadni?
- Hogyan kell fájlt paraméterben átadni?

Paraméterátadások

- Azt beszéltük meg, hogy a függvény paraméterei
  - A paraméterlistában vannak deklarálva
  - Az értéküket az aktuális paraméterekből kapják
  - Érvényességük és láthatóságuk a függvényre szorítkozik
  - Tehát az aktuális paraméter másolatáról van szó
- Következésképp a **getline(cin, s)** nem változtathatná meg **s** értékét, tehát sort nem is olvashatnánk be? Hogy lehetséges ez még is?

Paraméterátadások

- A különbség a paraméter átadás módjában van
- A C++ -ban kétfajta paraméterátadási mód létezik:
  - **Érték szerinti:** az eddig megismert mód, ilyenkor az aktuális paraméterről egy másolat készül és ezzel dolgozik a függvény. Az eredeti érték „védett” nem módosul.
  - **Referencia szerinti:** az aktuális paraméterre egy új referencia kerül deklarálásra. Azaz az eredeti lefoglalt memóriaterületet címkézzük fel újra.
    - Úgyis fogalmazhatunk, hogy az eredeti változót adjuk át (nem csak a másolatát), ezért a formális paraméteren végrehajtott változások az eredeti értéket módosítják.

Érték szerinti paraméterátadás

```
#include <iostream>
using namespace std;

void fv( double a ) {
    a = 0;
}

int main() {
    double d;
    d = 1;
    fv( d );
    cout << "eredmeny: " << d << endl;

    return 0;
}
```

Az eredmény: 1

Referencia szerinti paraméterátadás

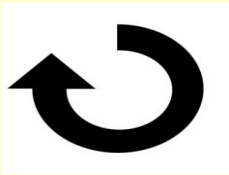
```
#include <iostream>
using namespace std;

void fv( double& a ) {
    a = 0;
}

int main() {
    double d;
    d = 1;
    fv( d );
    cout << "eredmeny: " << d << endl;

    return 0;
}
```

Az eredmény: 0

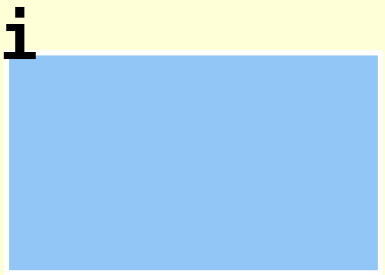
Referencia

- Lehetőségünk van olyan változókat létrehozni, amelyek egy másik változó értékét használják fel, azaz pontosan ugyanazt az értéket tárolják, mint a másik változó. Ezek az úgynevezett referenciák.
- A referencia önálló fogalom a C++ -ban
- A referenciát mindig egy már létező, tetszőleges változóra állítjuk rá, és ettől kezdve bármelyik értékének módosítása a másik értékét is módosítja, mivel ugyan arra a memóriaterületre fognak hivatkozni.
- A referencia típusának meg kell egyeznie az eredeti változó típusával, a neve pedig tetszőleges lehet
- A C++-ban a referenciaváltozókat az & operátorral jelöljük meg:
  - *<típus> <változó 1>;*
  - *<típus> & <változó 2> = <változó 1>;*

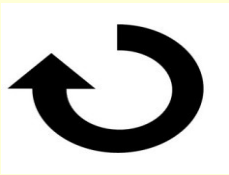
Változók I.



```
int i;   //deklaráció
```



Változók II.

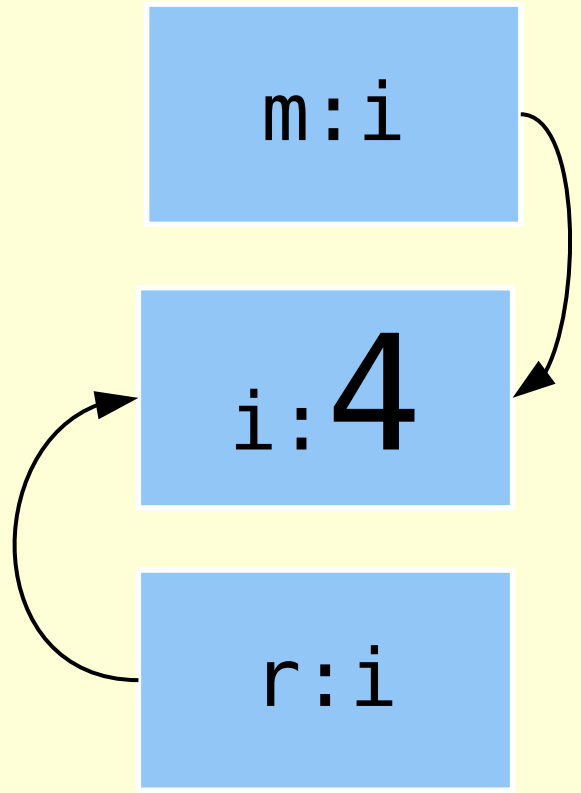


```
int i;  //deklaráció
i = 4;   //értékadás
```

i:4

referencia, mutató

```
int i;  //deklaráció
i = 4;  //értékadás
int &r = i;
int *m = &i;
```

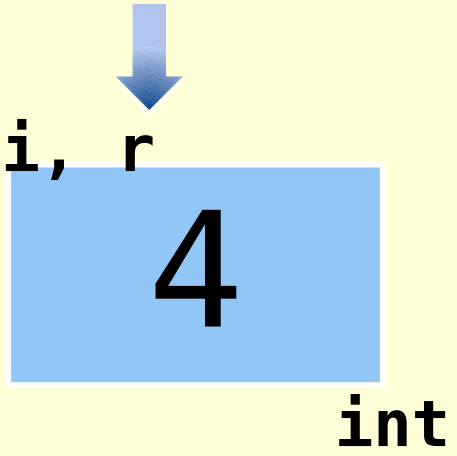


Referencia

- Referencia esetén nem mindig foglalódik le új memória terület, néha egy meglévőhöz rendelünk hozzá egy másik változónevet
- A memóriaterület mindkét névvel elérhető és változtatható.
  - `int i = 4;  // i=4;`
  - `int & r = i;  // i=r=4;`
  - `r = 10;  // i=r=10;`

Referencia egy értelmezése

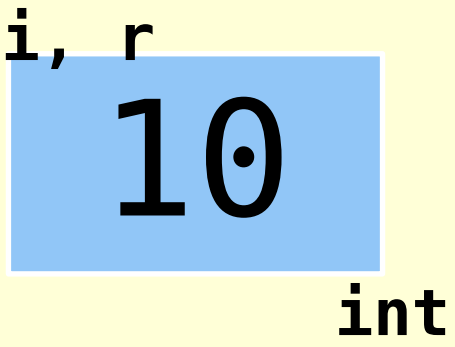
```
int& r = i;
```




```
// Új nevet rendelünk a
  már meglévő változóhoz
```

Referencia

```
int& r = i;
r = 10;
```



Referencia szerinti paraméterátadás

- Következmények:
  - ha szeretnénk, hogy egy függvény változtassa a változónkat, akkor lehetséges referencia szerint átvenni a paramétert
  - csak változó lehet a paraméter, kifejezés eredménye vagy konstans nem
  - gyors, mert nem kell másolatot készíteni, ami nagyobb memóriaigényű változóknál (pl. string egymillió karakterrel) lassíthatja a programot
  - veszélyes lehet, a legtöbb függvényhívásnál nem számítunk arra, hogy megváltozhat a paraméterül átadott változó (pl. matematikai függvények)

Konstans referencia szerinti paraméterátadás

- gyors, mert nem készül másolat
- biztonságos,mert nem változhat meg, figyel a fordítóprogram
- **const Típus& paraméter**
- legelterjedtebb paraméterátadási forma a gyakorlatban, ha nem alapípusokról van szó

```
int vektor_fuggveny2( const vector<int>& vek ) {
     sum += vek[0] //OK
     vek[0] = 0; //hiba!

}

int main() {
    vector<int> v(3,0);
    cout << vektor_fuggveny2( v ) << endl;
    return 0;
}
```

Több eredmény visszaadása

- Egyszerűen több paramétert veszünk át referencia szerint
  - Ezek értékével egyáltalán nem foglalkozunk, csak felülírjuk azokat
  - Így a paraméter jelentése az lesz, hogy „ide meg ide kérem az eredményt”
- Lesz még másfajta megoldás is erre a problémára
- Jövő héten megnézzük

Tömb, mint paraméter

```cpp
#include <iostream>
using namespace std;

int tomb_fuggveny1( int tomb[], int meret) {
    int osszeg = 0;
    for( int i = 0; i<meret; ++i){
        tomb[i] = (i+1)*3;
        osszeg += tomb[i];
    }
    return osszeg;
}

int main() {
    int t[3] = {0,0,0};
    cout << tomb_fuggveny1(t, 3) << endl;
    for( int i = 0; i<3; ++i ) {
        cout << t[i] << ", ";
    }
      return 0;
}
```

```cpp
#include <iostream>
using namespace std;

int tomb_fuggveny2( int* tomb, int meret) {
    int osszeg = 0;
    for( int i = 0; i<meret; ++i){
        tomb[i] = (i+1)*3;
        osszeg += tomb[i];
    }
    return osszeg;
}

int main() {
    int t[3] = {0,0,0};
    cout << tomb_fuggveny2(t, 3) << endl;
    for( int i = 0; i<3; ++i ) {
        cout << t[i] << ", ";
    }
      return 0;
}
```

Vektor, mint paraméter

- Az STL vector átadása paraméterként:
  - jelezzük a függvény paraméterlistájában, hogy milyen vektorra számítson
  - mivel ismeri a méretét, lekérdezhető (`.size()`), ezért itt már nem kell átadni ezt az értéket
  - szintaktika:
    - `int vektor_fuggveny1( vector<int> vek ){ … } //érték`

    illetve:
    - `int vektor_fuggveny2( vector<int>& vek ){ … } //referencia`

Tömb, mint paraméter

- az eredmény mindkét esetben:

```
Elemek osszege: 18
3, 6, 9,

Process returned 0 (0x0) execution time : 0.089s
Press any key to continue.
```

Vektor, mint paraméter

```cpp
#include <iostream>
#include <vector>
using namespace std;

int vektor_fuggveny1( vector<int> vek ) {
    int osszeg = 0;
    for( size_t i = 0; i<vek.size(); ++i){
        vek[i] = (i+1)*3;
        osszeg += vek[i];
    }
    return osszeg;
}

int main() {
    vector<int> v(3,0);
    cout << vektor_fuggveny1( v ) << endl;
    for( size_t i = 0; i<v.size(); ++i ) {
        cout << v[i] << ", ";
    }
    return 0;
}
```

```cpp
#include <iostream>
#include <vector>
using namespace std;

int vektor_fuggveny2( vector<int>& vek ) {
    int osszeg = 0;
    for( size_t i = 0; i<vek.size(); ++i ){
        vek[i] = (i+1)*3;
        osszeg += vek[i];
    }
    return osszeg;
}

int main() {
    vector<int> v(3,0);
    cout << vektor_fuggveny2( v ) << endl;
    for( size_t i = 0; i<v.size(); ++i ) {
        cout << v[i] << ", ";
    }
    return 0;
}
```

Vektor, mint paraméter

```
int vektor_fuggveny1( vector<int> vek ) { … }
```
esetén az eredmény:

```
Elemek osszege: 18
0, 0, 0,

Process returned 0 (0x0) execution time : 0.079s
Press any key to continue.
```

```
int vektor_fuggveny2( vector<int>& vek ) { … }
```
esetén az eredmény:

```
Elemek osszege: 18
3, 6, 9,

Process returned 0 (0x0) execution time : 0.079s
Press any key to continue.
```

Tömb / vektor, mint paraméter

- **Primitív tömb** esetén mind a szintaktikai különbség nem takar funkcionális különbséget: a függvények az eredeti tömbbel fognak dolgozni, a változások ezért maradandóak, mivel a tömb pointerrel van megvalósítva. (Vagyis tömböt nem tudunk úgy paraméterben átadni, hogy másolat készüljön róla, a mutató érték szerint átadva ugyanoda mutat ahová az eredeti mutató).
- **Vektor** esetén a változóknál megszokott érték illetve referencia szerinti paraméterátadást tapasztaljuk.
  - Első esetben a vektorunkról másolat készül, a műveleteket ezen a másolaton hajtjuk végre, az eredeti vektorunk változatlan marad.
  - Második esetben az eredeti vektorra hivatkozunk egy új referenciával, nem készül másolat a műveletek az eredeti vektort módosítják, ez maradandó.

Fájlok paraméterben

- A fájlokat (ifstream, ofstream) illetve bármilyen csatornát (iostream) mindig referencia szerint kell átvenni

- **Miért?** – Mert nem készíthető róluk másolat, aminek azaz oka, hogy például a „hol tartunk a fájlban” benne van a fájl típusban:
  - képzeljük el, hogy egy függvény lemásol (érték szerint átvesz) egy ifstream-et és olvas belőle
  - visszatérés után a következő olvasásnál a fájl előző olvasásainak kellene újra megtörténnie, hisz csak a másolatból olvastunk, az átadottból nem, az tehát nem is mehet arrébb
  - ez viszont követhetetlen, illetve technikailag rémálom lenne a megvalósítása

Fájlok paraméterben

```
#include <iostream>
#include <fstream>
using namespace std;

void olvas( ifstream& f, double& a ) {
    f >> a;
}

int main() {
    double d;
    ifstream befile( "a.txt");
    olvas( befile, d );
    cout << "eredmeny: " << d << endl;

    return 0;
}
```

Összefoglalás

- Paramétert kétféleképpen is át lehet adni C++ -ban
  - Érték szerint: másolat készül egy lokális változóra
  - Referencia szerint: ugyan az a változó több néven
- Paraméterként a fájlok és a tömbök speciálisak
- Mivel veszélyes (felülírás!) mindent referenciaként átvenni, ezért csak akkor tegyük, ha
  - a specifikációnk szerint eredményt adunk vissza benne
  - → konstans referenciát használjunk inkább ha hatékonysági okokból szeretnénk csak referenciát