

6. Gyakorlat

Rövid elméleti összefoglaló

- **Függvények deklarációja és definíciója**

A saját függvényeinket mindig definiálni kell. A definíció, amelyet csak egyszer lehet megadni, a C++ programon belül bárhol elhelyezkedhet. Ha a függvény definíciója megelőzi a függvény hívásának helyét, akkor ez helyettesíti a függvény prototípusának külön megadását. A prototípus: `<visszatérési típus> Függvénynev(paraméterek típusa);`

<pre><visszatérési típus> Függvénynev(paraméterek) { //függvény törzse <lokális definíciók és deklarációk> <utasítások> }</pre>	<p>Ha a visszatérési típus void, akkor a függvénynek nem kell tartalmaznia return utasítást, mivel nem ad vissza semmit.</p> <p>Ellenkező esetben a return melletti kifejezés típusának meg kell egyeznie a függvény visszatérési típusával. Ebben az esetben a függvénynek ez a visszatérési értéke, vagyis a függvény működésének eredménye.</p>
---	---

A függvény paraméterlistáján a paraméterek megadása: a paraméter típusa a paraméter nevével (vesszővel tagolva, ha több paraméter is van), például: `int n`, `double c` stb. A paraméterlistán adjuk meg a bemenő, és a kimenő paramétereket egyaránt. A bemenő paramétereknek kell, hogy értékük legyen, a kimenő paraméterek a függvény törzsében kapnak értéket. Csak pointerrel, vagy referenciával megadott paraméter segítségével adhatunk ki értéket.

<pre>int f1(int a, int b) { int c; c = a+b; // összead return c; }</pre>	<p>Az <code>f1()</code> függvénynek két egész típusú bemenő paramétere van: az <code>a</code> és a <code>b</code>. A függvény feladata, hogy a két bemenő paraméter összegét adja vissza visszatérési értéként.</p> <p>Lokálisan definiálunk egy egész típusú <code>c</code> változót. A <code>c</code> változóba adjuk össze az <code>a</code> és a <code>b</code> értékeket, majd a return utasítással adjuk vissza a <code>c</code> tartalmát.</p>
<pre>int f1(int a, int b) { return a+b; }</pre>	<p>A feladatot így is megoldhatjuk:</p> <p>Az egyszerűbb számítást tartalmazó kifejezést a return mellett közvetlenül is megadhatjuk.</p>

- **Inline függvények**

Az **inline** („soron belüli”) függvényekkel helyettesíthetjük a **#define** makrókat. Az **inline** függvény esetén a függvény törzsét képező kód helyettesítődik be a függvényhívás helyére („kódmakro”). Általában kisméretű, gyakran hívott függvények esetén ajánlott alkalmazni ezt a megoldást. Az **inline** függvények használatának előnye, hogy a függvényhíváskor az argumentumok feldolgozása teljes körű típusellenőrzés mellett megy végbe.

<pre>//MINTA6_01 #include <iostream> using namespace std; // inline függvény definiálása inline int min(int a, int b) { return a < b ? a : b; } int main() { int x = 13, y = 5, w; w = min(x,y); cout << "A kisebb adat : " << w << endl; cin.get(); return 0; }</pre>	<p>Írjunk inline függvényt, amely két egész szám közül kiválasztja a kisebbet!</p> <p>A <code>min()</code> inline függvény aktiválása.</p> <p>A program futásának eredménye:</p> <p>A kisebb adat : 5</p>
---	---

- **Mutató típusú paraméter**

Kimenő paramétert a megfelelő típusra mutató pointerrel adhatjuk meg. Ez azonban nem jelenti azt, hogy bemenő paraméter nem lehet pointeres változó. A C++ nyelvben biztonságosabb a referencia típus használata.

- **Referencia típusú paraméter**

A referencia típus igazi lehetőségét a hivatkozás szerinti paraméterátadás jelenti. (A háttérben a fordító valójában címeket másol, azonban a forrásprogramból ez nem látható.) A kimenő paramétereket referenciaként adjuk meg.

Mindkét változatra nézzünk meg egy-egy egyszerű függvényt!

<pre>void f2(int a, int b, int& c) { c = a*b; // szorzat }</pre>	<pre>void f3(int a, int b, int* c) { *c = a - b; //különbség }</pre>
--	--

Az $f_2()$ függvény a szorzat eredményét referenciaként megadott c paraméterben adja vissza, míg az $f_3()$ függvény a különbséget az int -re mutató c pointerrel kijelölt tárolóban szolgáltatja vissza. A függvények definiálását és hívását a *MINTA6_02* mintafeladatban láthatjuk.

```
//MINTA6_02
#include <iostream>
using namespace std;
// bemenő paraméter: a , b
// kimenő érték a return mellett a visszatérési érték
int f1(int a, int b) // definíció és a prototípus egyben
{
    int c; // lokális definíció
    c = a+b; // összeadás
    return c;
}
//elhagyható a paraméter neve a prototípus megadásánál
void f2(int, int, int&);
void f3(int a, int b, int* c); // prototípus
int main()
{
    int x = 6, y = 3, osszeg, szorzat, kulonbseg;
    osszeg = f1(x, y); // f1 függvény hívása
    f2(x, y, szorzat); // f2 függvény hívása
    f3(x, y, &kulonbseg); // f3 függvény hívása
    cout << x << " * " << y << " = " << szorzat<< endl;
    cout << x << " + " << y << " = " << osszeg<< endl;
    cout << x << " - " << y << " = " << kulonbseg<< endl;
    cin.get();
    return 0;
}
// bemenő paraméter: a, b
// kimenő paraméter a c (referencia)
void f2(int a, int b, int& c) // f2 függvény definíciója
{
    c = a*b; //szorzat
}
// bemenő paraméter: a, b
// kimenő paraméter a c által mutatott tároló (*c)
void f3(int a, int b, int* c) // f3 függvény definíciója
{
    *c = a - b; //különbség
}
```

Javasolt programozási módszer:

- A függvények prototípusát a **main()** függvény előtt adjuk meg.
- A függvényeket a **main()** függvény után dogozzuk ki.

Megjegyzések a main() függvényhez:

Az x és y változóknak már a definiálásnál kezdőértéket adunk.

Az $f_1()$ függvény bemenő argumentumai az x és az y változók, az eredmény az *osszeg* változóba másolódik.

Az $f_2()$ függvény **void** típusú, tehát nincs visszatérési értéke. Hívásakor az x , y a bemenő argumentumok, melyeknek van értékük, a *szorzat* változót azonban átveszi a függvény és beleírja az eredményt, azaz a két paraméter szorzatát.

Az $f_3()$ függvény szintén **void** típusú, tehát itt sincs visszatérési érték. A bemenő argumentumok az x és az y változók értéke. Az **int*** c paramétert a *kulonbseg* változó címével helyettesítettük (így kapjuk meg az eredményt).

A program futásának eredménye:

```
6 * 3 = 18
6 + 3 = 9
6 - 3 = 3
```

Számítsuk ki két valós adat számtani és mértani közepét függvénnyel, a kimenő paraméternél használjunk referenciát!

```
//MINTA6_03
#include <iostream>
using namespace std;
void SzamtaniKozep( double a, double b, double& szk);
void MertaniKozep( double a, double b, double& mk);
int main()
{
    double x = 1, y = 2, szkopez, mkozep;
    SzamtaniKozep(x,y,skopez);
    MertaniKozep(x,y,mkozep);
    cout << "Szamtani kozep: " << szkopez << endl;
    cout << "Mertani kozep : " << mkozep << endl;
    cin.get();
    return 0;
}
void SzamtaniKozep( double a, double b, double& szk)
{
    szk = (a + b)/2;
}
void MertaniKozep( double a, double b, double& mk)
{
    mk = sqrt(a * b);
}
```

Megjegyzés:

- Pointer helyett használjunk referenciát:
int *p helyett **int &p**
int **pp helyett **int *&pp**
- A referenciához nem készíthetünk
 - mutatót (**int &***) és
 - referenciát (**int &&**).

A program futásának eredménye:

```
Szamtani kozep: 1.5
Mertani kozep : 1.41421
```

• Tömbök átadása függvénynek

A tömböt nem lehet érték szerint átadni a függvénynek, illetve függvényértékként megkapni. Az **int[]** típusú vektorargumentum **int*** típusú mutatóként adódik át a függvénynek. A függvényen belül a tömbben végrehajtott változások a függvényen kívül is megmaradnak.

```
//MINTA6_04
#include <iostream>
using namespace std;
void TombOlvas(int y[], int n);
void OsszegSzamol(int y[], int n, int& ossz);

int main()
{
    int x[10], db, osszeg = 0;
    db = 4; // a tömbnek 4 eleme lesz
    TombOlvas(x,db);
    OsszegSzamol(x,db,osszeg);
    cout << "A tomb elemeinek osszege: "
         << osszeg << endl;
    cin.get(); cin.get();
    return 0;
}

void TombOlvas(int y[], int n)
{
    for (int i = 0; i<n; i++)
    {
        cout << i << ".elem : "; cin >> y[i];
    }
}

void OsszegSzamol(int y[], int n, int& ossz)
{
    ossz = 0;
    for (int i = 0; i<n; i++)
        ossz += y[i];
}
```

A *TombOlvas()* függvény feladata a paraméterként kapott tömb adott darabszámú adattal való feltöltése.

Az *OsszegSzamol()* függvény a tömb elemeinek összegét számítja ki, amit a paraméterlistán ad vissza.

A függvények aktiválása argumentumukkal.

Az eredmény kiírása.

A *TombOlvas()* függvény feladata:

az *y* tömb elemeinek feltöltése adatokkal

Az *OsszegSzamol()* függvény a tömb (*y*) és a tömb elemeinek száma (*n*) ismeretében kiszámítja a tömb elemeinek összegét az *ossz* referencia-paraméterbe.

A program futásának eredménye:

```
0.elem : 4
1.elem : 1
2.elem : -2
3.elem : 0
A tomb elemeinek osszege: 3
```

• Struktúra átadása függvénynek

Ha a függvényben a struktúra bármelyik adattagján változtatni szeretnénk, akkor a struktúrát referenciaként kell átadnunk, például: *muvelet& v*, vagy a mutatójával.

```
//MINTA6_05
#include <iostream>
#include <cstring>
struct muvelet{
    double a,b; // a két operandus
    double ered; // az eredményt tárolja
    char muvjel; // műveleti jel
};

// A függvények prototípusai
void Feltolt(muvelet& v);
void Szamol(muvelet& v);

int main()
{
    muvelet m;
    Feltolt(m); Szamol(m);
    cout << m.a << " " << m.muvjel << " " << m.b << " = ";
    cout << m.ered;
    cin.get(); cin.get();
    return 0;
}
```

A *muvelet* struktúra *a* és a *b* adattagja a két operandus, a *muvjel* a műveleti jelet tartalmazza, és az eredményt pedig az *ered* adattagja tárolja.

Megjegyzések a *main()* függvényhez:

A *Feltolt()* függvényt aktiváljuk az *m* változóval, amely egy *muvelet* típusú struktúra. Ez a függvény ad értéket az *a,b*, és a *muvjel* adattagoknak. Ha a műveleti jel / (osztás), akkor a *b* adattag nem lehet 0, vizsgálat a beolvasáskor.

A *Szamol()* függvény az adatokkal feltöltött *muvelet* típusú *m* struktúra változót módosítja a kiszámított eredménnyel, majd az eredményeket kiírjuk.

<pre> void Feltolt(muvelet& v) { do { cout << "Muveleti jel: "; cin >> v.muvjel; }while (!strchr("+-*/",v.muvjel)); cout << "1. adat: "; cin >> v.a; do { cout << "2. adat: "; cin >> v.b; }while (v.muvjel == '/' && v.b == 0); } void Szamol(muvelet& v) { switch (v.muvjel) { case '+': v.ered = v.a + v.b; break; case '-': v.ered = v.a - v.b; break; case '*': v.ered = v.a * v.b; break; case '/': v.ered = v.a / v.b; break; } } </pre>	<p>A <i>Feltolt()</i> függvény feladata:</p> <p>A <i>v</i> paraméter adattagjainak beolvasása. Ha a műveleti jel / azaz osztás, akkor a <i>b</i> adattag nem lehet 0, vizsgálat a beolvasáskor.</p> <p>A <i>Szamol()</i> függvény feladata:</p> <p>A műveleti jel alapján a <i>v</i> paraméter <i>ered</i> tagjába kerül az eredmény a megfelelő művelet kiszámítása után.</p> <p>A program futásának eredménye:</p> <pre> Muveleti jel: * 1. adat: 4 2. adat: 6 4 * 6 = 24 </pre>
--	--

• Függvényre mutató pointer átadása függvénynek

<pre> //MINTA6_06 #include <iostream> #include <iomanip> #include <cmath> using namespace std; typedef double fv(double x); typedef fv *pfv; double fgl (double x); void tablazat(double xk, double xv, double dx, pfv f); int main() { double kezdo=0, veg=95, lepeskoz = 10; tablazat(kezdo, veg, lepeskoz, fgl); cin.get(); return 0; } double fgl (double x) { double rad, y; rad = x*3.141592654/180; y = sin(rad); return y; } void tablazat(double xk, double xv, double dx, pfv f) { double xx, yy; xx = xk; cout.setf(ios::showpoint); cout.precision(4); cout << " x" << " sin(x)" << endl; while (xx < xv) { yy = (*f)(xx); // (*f)(xx); azonos a f(xx) hívással cout << " " <<xx << " " << yy << endl; xx += dx; } } </pre>	<p><i>fv</i> olyan függvény, amelynek paramétere double és visszatérési értéke double <i>pfv</i> az <i>fv</i>-re mutató pointer típusa Az <i>fgl()</i> függvény bemenő paramétere a szög értéke fokban, visszatérési érték a radiánban átalakított szög szinusza. A <i>tablazat()</i> függvény bemenő paramétere <i>xk</i> a kezdőérték, az <i>xv</i> a végérték, a <i>dx</i> a lépésköz, <i>f</i> a függvényre mutató pointer.</p> <p>Az adatok kezdőértékként való megadása. A <i>tablazat()</i> függvény aktiválása.</p> <p><i>fgl()</i> függvény: bemenő paraméter fokban. Feladata a fok átszámítása radiánba a sin függvény számára. A sin <i>x</i> értékének számítása a visszatérési értékként való megadása, amely érték egy utasításként is megadható: return sin(x*3.141592654/180);</p> <p>Táblázat készítése <i>xk</i> kezdőértéktől <i>xk</i> végértékig, <i>dx</i> lépésközzel.</p> <p>A táblázandó függvényt paraméterként kapja. Az <i>f</i> a függvényre mutató pointer. Az <i>xx</i> bemenő paraméterből kiszámítja az <i>yy</i> visszatérési értéket.</p> <p>A program futásának eredménye:</p> <pre> x sin(x) 0.000 0.000 10.00 0.1736 20.00 0.3420 30.00 0.5000 40.00 0.6428 50.00 0.7660 60.00 0.8660 70.00 0.9397 80.00 0.9848 90.00 1.000 </pre>
--	---

Feladatok

1. Tervezzünk **inline** függvényt két pont távolságának meghatározására, írjunk hozzá **main()** függvényt! (GYAK6_1)

```
inline double tav(int x1,int y1,int x2,int y2)
```

A program futásának eredménye:

```
x1: 1
y1: 1
x2: 2
y2: 2
Tavolsag: 1.41421
```

2. Használjuk az alábbi definíciót és deklarációt! Írjunk függvényt a henger adatainak beolvasására, valamint a henger felszínének és térfogatának kiszámítására! Aktiváljuk a függvényeket a **main()** függvényben, és jelenítsük meg az eredményeket! (GYAK6_2)

```
void OlvasHengerAdatok(double& r, double& h);
void FelszTerf(double r, double h, double& felszin,
               double& terfogat);

double sugar, magassag, felsz, terf;

henger felszíne:  $2 \cdot r^2 \cdot \pi + 2 \cdot r \cdot \pi \cdot h$ 
henger térfogata:  $r^2 \cdot \pi \cdot h$ 
```

A program futásának eredménye:

```
Henger sugara      : 1
Henger magassaga: 1

A henger felszine : 12.5664
A henger terfogata: 3.14159
```

3. Használjuk az alábbi definíciót és deklarációt!

```
const double PI = 3.141592654;
struct henger{
    double r, h, felsz, terf;
};

void OlvasHenger(henger& v);
void Szamol(henger& v);
henger h;
```

Írjunk *OlvasHenger()* függvényt, amely a henger adatait olvassa be; *Szamol()* függvényt, amely kiszámítja a henger felszínét és térfogatát! Mindkét függvényt aktiváljuk a **main()** függvényben, és az eredményeket jelenítsük meg. (GYAK6_3)

4. Használjuk fel az alábbi prototípusokat, és definíciókat!

```
void VektorOlvas(double y[], int *n);
void PozitivAtlag(double y[], int n, double& pozatlag);
void VektorKiir(double y[], int n);
int main()
{
    double x[100], pozatl;
    int elemszam;
    . . .
}
```

A *VektorOlvas()* függvényben kérdezzünk rá az aktuális adatok számára (*n), amely nem haladhatja meg a 100 értéket, majd töltsük fel az y tömböt valós adatokkal!

A *PozitivAtlag()* függvényben az y tömb és az n elemszám ismeretében számítsuk ki a tömb pozitív elemeinek átlagát, melyet a referenciaként átvett *pozatlag* paraméterben adunk vissza!

A *VektorKiir()* függvényben jelenítsük meg az y tömb elemeit!

Írjuk meg a függvényeket és tervezzünk hozzá **main()** függvényt, aktiváljuk a függvényeket, majd írjuk ki a *pozatlag* eredményt! (GYAK6_4)

A program futásának eredménye:

A tömb elemeinek száma (1	A tömb elemei:
A tömb elemei:	12
0 : 12	-4
1 : -4	0
2 : 0	6
3 : 6	-2
4 : -2	Pozitív elemek atlaga: 9

5. Használjuk fel az alábbi prototípusokat és definíciókat!

```
typedef int vekt[10];
void TombOlvas(vekt y, int n);
void OsszegSzamol(vekt y, int n, int& ossz);
int main()
{ int db, osszeg = 0;
  vekt x;
  db = 4; // 4 elemű lesz az x tömb
  . . .
}
```

A *vekt* felhasználói típus olyan tömböt jelent, melynek 0-9 indexelhető, 10 darab egész értékeket tároló eleme van.

A *TombOlvas()* függvényben az *n* paraméterben megadott elemszámú, *vekt* típusú *y* tömb elemeinek adjunk egész típusú értékeket!

Az *OsszegSzamol()* függvényben az *n* elemszámú *y* tömb elemeinek számítsuk ki az összegét, és azt a referenciaként átvett *ossz* paraméterben adjuk vissza!

Írjuk meg a függvényeket, tervezzünk hozzájuk *main()* függvényt, majd jelenítsük meg az eredményeket! (GYAK6_5)

A program futásának eredménye:

```
0.elem : -2
1.elem : 5
2.elem : 3
3.elem : 4
A tömb elemeinek osszege: 10
```

6. Használjuk fel az alábbi prototípusokat és definíciókat!

```
int* TombOlvas( int* n);                int db, osszeg = 0;
void OsszegSzamol(int* y, int n, int& ossz);  int *x;
```

A *TombOlvas()* függvényben olvassuk be a tömb elemeinek darabszámát, melyet a paraméterlistán adunk vissza. Foglaljunk helyet a dinamikus tömbnek, adjunk értéket az elemeinek, majd függvényértékként adjuk vissza a tömb kezdő-címét.

```
int* TombOlvas(int* n)
{ int* y;
  cout << "Elemek száma: "; cin >> *n;
  y = new int[*n];
  for (int i = 0; i<*n; i++)
  {
    cout << i << ".elem : "; cin >> y[i];
  }
  return y;
}
```

Az *OsszegSzamol()* függvényben összegezzük, és adjuk vissza az összeget az *ossz* paraméterben!

Írjuk meg a függvényeket, és tervezzünk hozzájuk *main()* függvényt, írassuk ki az eredményeket! (GYAK6_6)

A program futásának eredménye:

```
Elemek száma: 5
0.elem : 12
1.elem : -6
2.elem : 0
3.elem : 4
4.elem : 2
A tömb elemeinek osszege: 12
```