# Objektumorientált programozás a C++ segítségével

## **Bevezető**

## A C nyelv és egy program felépítése

A C nyelv a C++ előfutára, a ’60-as évek végén fejlesztette Dennis Ritchie és Bjarne Stoustroup. Eleinte adatbáziskezelési célból írták, de később robusztussága miatt széleskörű használatúvá vált. Segítségével könnyen konfigurálható, egyszerűen vezérelhető programokat lehet írni.

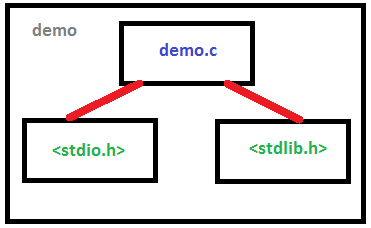
A **C programok** (.c) rendszerint egymodulosak (monolitikusak), melyekhez testreszabható könyvtárakat, headereket (.h) lehet rendelni. A **headerek** újrafelhasználható kódokat tartalmaznak, melyeket tetszés szerint lehet egymás között meghivatkozni, ösztönözve a kisméretű, könnyen szerkeszthető modulok írására.

Fontos megjegyezni, hogy a headerek rendszerint nem konkrét programpéldányokat futtatnak, hanem minimálisan megvalósított programlogika leírásait tartalmazzák. A .c program felelős a fő programlogika megszervezéséért és kezeléséért.

Programok összerendeléséhez és –fordításához egy fejlesztőkörnyezetet (IDE) használunk, pl. **Code::Blocks**. Az IDE képes összelinkelni a forrásfájlokat, majd gépi kóddá alakítva azt, futtatható állományként indítani azt.

A fejlesztőkörnyezet két részből áll:

* **interpreter**: programutasításokat értelmező rutin. Az interpreternek megadhatunk ún. *preprocesszori direktívákat/makrókat*, melyek a program futtatásakor elsőbbséget élveznek, más szóval előbb értékelődnek ki a normál programutasításoknál.
* **compiler**: kész programot gépi kóddá fordító rutin. A compiler végzi a különféle forrásfájlok összerendelését (linking), melyek a projekthez tartoznak. Minden projekthez tartozhat debug és release verzió, melyek önállóan is futtathatók.



**Példa 1.: a demo projekt és felhasznált fájljai**

## Kódolás C-ben

Egy C program szerkezete nagyon egyszerű, két részre bontható:

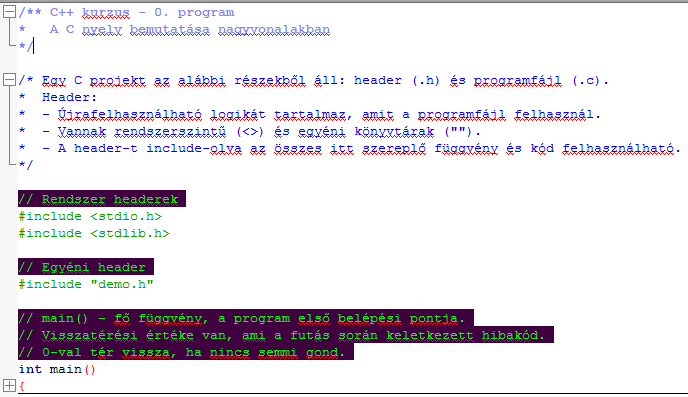
* **header- és makrófelsorolások**: különféle fordítói direktívákat tartalmazó rész. Itt végezhetők el a headerek „behúzásai”, illetve a makrók.
* **függvények és utasítások**: a tényleges programlogikát tartalmazó részek. Ide tartoznak a változók és deklarációjuk, programobjektumok, függvényhívások.

A C fordító az utasításokat **szekvenciálisan**, azaz leírásuk sorrendjében hajtja végre. A kódhoz fűzhetünk **kommenteket**, amikkel nehézkesebb vagy bonyolultan megérthető kódrészletek viselkedését, működését magyarázhatjuk meg.

A C egy **erősen típusos** nyelv, ami azt jelenti, hogy a programban tárolt változók típusa előre lekötött, és jellege számít kiértékelésénél elágazásokban, feltételekben és más programszerkezetekben. A nyelvben lehetőség van használni az előre megírt (**primitív**) típusokat, illetve sajátot is hozhatunk létre (ennek az eszköze pl. a struktúra, lásd később).

Más nyelvektől eltérően, a C képes bizonyos szinten közvetlen a memóriában is műveleteket végezni. Ebből a célból hozták létre a **pointert**, egy változótípust, mely képes a memória egy helyére (cím), adott méretű területre mutatni.

Ahogy az előző pontban tárgyaltuk, a fordító képes több header- és programfájlt összefordítani. Ezek közül egyikőjükben fel kell sorolni a **main()** függvényt, mint a program fő belépési pointját. A main() egy int típusú függvény, melynek visszatérési értéke tükrözi az alkalmazás visszaadott hibakódját. Ha nincs semmi gond, 0 értékkel térünk vissza.



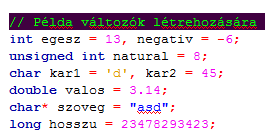
**Példa 2.: a demo programunk felépítése: headerek, main() és kommentek elhelyezve**

## Primitív típusok

A primitív típusok nagyrésze más nyelvekből már jól ismert adattagokat kezel. Ezeknek különböző értékkészlete és adottsága van, amiket mindig szem előtt kell tartani, amikor használatukkal élünk.

Különböző változatok:

* **int, egész**: egész szám tárolására képes típus. Egész szám lévén, tizedesjegyet nem tárol. Operációs rendszertől függően 32/64 bites számokat tárolhatunk egy változóban. Amennyiben előjel nélküli (természetes) számot tárolunk, akkor az **unsigned** szót a típusdeklaráció elé írhatjuk.
* **double, valós**: lebegőpontos számokat tároló típus. Előforduló számaink legtöbbjét ezzel a típussal jellemezhetjük, ha statisztikai vagy bonyolultabb matematikai műveleteket alkalmazunk.
* **char, karakter**: egy karakter (betű, szimbólum) tárolására képes típus. Egy karakter meghivatkozható a jelével aposztrófok között (’C’), illetve a helyettesítő kódjával a kódtáblából (ASCII-kód, pl. C-nél 67). A fordító mindig a kóddal dolgozik (és nem a betűvel, de át tudja azt fordítani értelemszerűen), ezért azon úgy végezhetők műveletek, mint egy egész esetében.
* **string – char\*, szöveg**: karakterek egymásutánja, egy lezáró karakterrel (nulla, ”\0”) a végén. A C-beli szöveg adottsága, hogy a lezáró nulla miatt mindig eggyel több a hossza, mint a tényleges szöveghossz. Karaktereit külön-külön elérhetjük a [] operátor segítségével, 0-tól kezdődően. Magát a szöveget idézőjelek (””) közé tesszük deklarációkor, lezáró 0 nélkül.



**Példa 3.: változók deklarálásának menete**

## Advankamentált típusok – tömb

A **tömb** egy azonos típusú értékek tárolására alkalmas konstrukció. Jellegzetessége, hogy mindig méretével együtt adjuk meg, ami később nem változtatható meg. A tömbelemeket mindig a tároló nevével hivatkozzuk meg, plusz az indexelő operátor (**[]**) segítségével, mely az elem sorszámát kapja értékül.

A tömb első elemét konvenció szerint 0 értékkel helyettesítjük, így a szó szerinti második elem az 1-es, aztán 2-es, stb. Minden tömb előre meghatározottan fel is tölthető, {} között az elemfelsorolások vesszővel elválasztva.



**Példa 4.: tömb használata**

## Advankamentált típusok – pointer

A **pointer** egy nagyon erős fegyver, és a C nyelvcsalád egyik legrettegettebb eszköze.

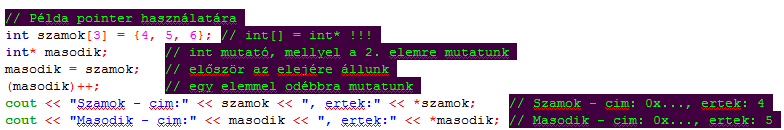
Mutatók segítségével módunk nyílik arra, hogy a számítógép memóriájában közvetlenül hozzuk létre és menedzseljük a változóinkat. Ezt a válfajt **dinamikus memóriakezelésnek** nevezzük, és szabad kezet ad a programozónak az értékkezeléshez. Eredményképpen nagyban javíthatunk a program hatékonyságán, mikor is elkerülhetjük a primitív tömb okozta memóriakorlátokat.

Maga a mutató nem más, mint a memória egy szeletére (blokkjára) hivatkozás. A mutató mindig egy hexadecimális értéket ad vissza, viszont azt is megtehetjük vele, hogy az általa mutatott értéket kérjük le. A mutatón továbbá úgy végezhetők az összeadás-kivonás műveletek, mintha adott számú típusnyit mennénk odébb.

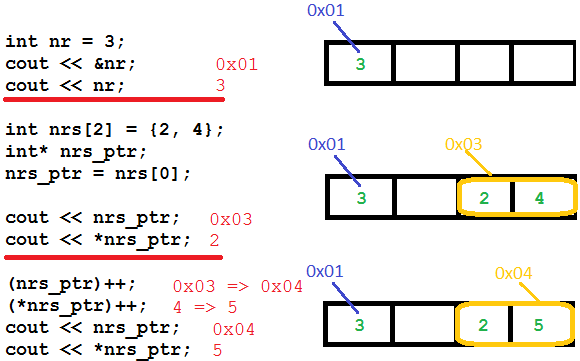
A pointer talán leghasznosabb tulajdonsága, hogy minden változón értelmezhető, igaz, külön szintaktika érvényes rá (C++ értelem):

|  |  |  |
| --- | --- | --- |
|  | **pointer (<type>\*) esetén** | **változó (<type>) esetén** |
| **&** - címképző operátor | pointer címe | változó címét adja vissza |
| **semmi** operátor | pointer címe | változó értékét adja vissza |
| **\* -** dereferencia operátor | pointer hivatkozott értéke | nincs értelme |

Érdekesség: a tömb és a pointer félig-meddig kölcsönösen egyértelműek. A tömb pointerként való használata a tömb első elemét címzi meg és adja vissza.



**Példa 5.1: pár pointer típuson végzett művelet**



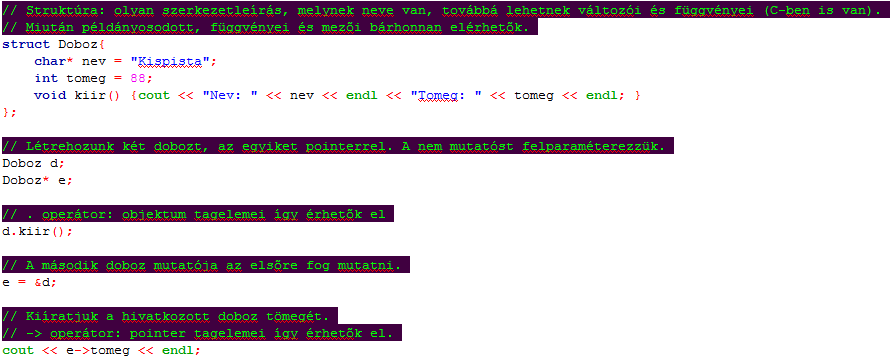
**Példa 5.2: egyszerű példa, mely bemutatja a lehetséges műveleteket**

## Advankamentált típusok – struktúra

A **struktúra** egy összetett csoport, mely képes függvények és változók csoportba foglalására. Egy struktúra egyféle mini-programként viselkedik, mely kis méretűtől nagyon nagyig is terjedhet.

Alapesetben objektumként viselkedik, viszont pointerként is inicializálhatjuk. Pointerként rendszerint akkor szoktuk használni, amikor vissza kell adni egy újonnan létrehozott objektum címét például.

Fontos megjegyezni, hogy objektum és pointer tagelemeit más-más operátorral érjük el.



**Példa 6.: struktúra használata pointerrel és objektumként**

A struct egyik gyakorlati megvalósítása matematikai konstrukciókkal történhet. Nehézséget okozhat pointerek tárolása egy kalap alatt, például több Doboz tárolása egy helyen, mivel primitív tömbnek nem adhatunk elemül Doboz pointert. Ennek a kiküszöbölésére találták ki a **láncolt listá**t.

A láncolt lista egy a programban (kód szerint) nem létező kompozíció, mely több ugyanolyan típusú elemet tartalmaz. Minden elem az alábbiak szerint épül fel:

struct Elem {

<típus> value = érték; //a ténylegesen tárolt elem értéke

Elem\* next; //a listában következő elem referenciája / címe

};

A láncolt lista jellegéből adódik, hogy a listaelemeket nem kell összefüggően lefoglalni a memóriában, hanem azok tetszőlegesen **lehetnek szétszórva**. Arra kell figyelni, hogy a következő elemnek a mutatója érvényes legyen, különben nagy bajban találhatjuk magunkat! ☺

Navigálni egy láncolt listában egyszerű: a jelenlegitől n távolságra lévő elemet úgy érhetjük el, hogy egy számlálós ciklusban nyilvántartunk egy ideiglenes Elem\* pointert, aminek a next paraméterét minden iterációban meghívjuk és eltároljuk. Lehetséges ugyanezt elérni **rekurzióval**, függvények egymás után hívásával, ami mindig az aktuális objektum next paraméterével tér vissza elegendő számú hívás után.

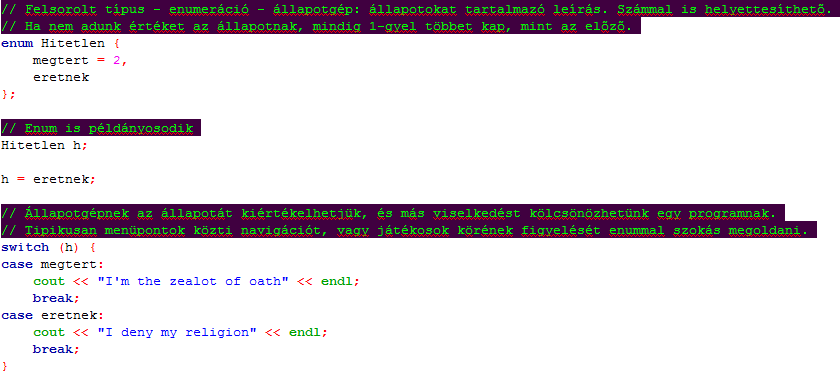
A láncolt listának továbbfejlesztése lehet a **kétirányúsítás,** melynél minden elemnél egyúttal az előző elem referenciáját is eltároljuk. Segíthet az is, ha minden elemnél eltároljuk az első elemet, és így tovább - a struktúrák bonyolításával a különféle tömbműveletek egyszerűsödhetnek, de bonyolódhatnak is.

## Advankamentált típusok – felsorolt típus

A **felsorolt típus** (állapotgép, enumeráció) egy tömbhöz hasonló egyszerű szerkezet, mely állapotokat tartalmaz és valamelyiküket mindig aktívként tartja számon.

Az enum állapota tetszőlegesen frissíthető a program bármely pontján, viszont új állapotokat nem fűzhetünk hozzá, illetve meglevőket nem vehetünk el. Egy állapotgép tetszőleges számú példánnyal rendelkezhet, illetve több különböző állapotgép is létezhet egyszerre.

Az egyes állapotok egyedi indexszel rendelkeznek, mint egy tömb elemei is. Ezeket felülírhatjuk a definícióban – ha egy számértékes elem után nem írjuk ki a következő elemek azonosítóját, akkor azok mindig a soron következőt kapják (alábbi példa esetén Hitetlen 🡪 eretnek állapota 3-as lesz).

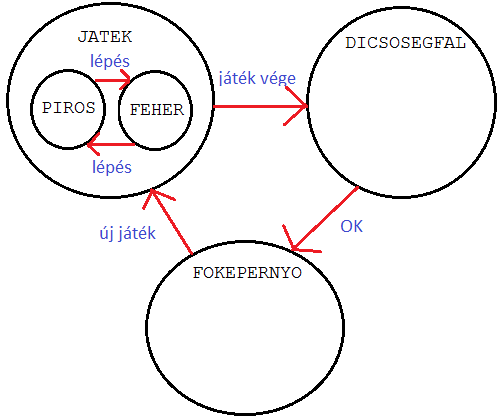


**Példa 7.1: egy kultista lehet megtért és eretnek. Állapottól függően mást mond.**

Gyakorlatban egy enummal lehet a program **futásának állapotait** jellemezni. Tekintsünk egy játékot, mely navigál egy menürendszerrel. Ha a menürendszert akarjuk modellezni, akkor a menüpontokat szemléltethetjük egy állapotgép állapotaival: játék, töltés, mentés, kilépés, dicsőségfal, beállítások.

Minden állapotban végezhetünk egy kirajzolást és egy egyéni viselkedést, melyekből bizonyos eseményekre a program más állapotokba ugrik, netalán marad ugyanabban az állapotban.

Tekintsük a fenti példa egy lehetséges állapotgépét, egy játék menetét:



**Példa 7.2: Egy játék állapot egy egyszerű játékban, ahol piros-fehér felváltva lép**

## **Az OOP – vezérlőelv a strukturált programozáshoz**

Az objektumorientált programozás egyik legmeghatározóbb elve korunk fejlesztési eljárásainak, mint pl. a Szent Grál is volt Arthur király eszköztárában. Programjaink nagy többségét önálló viselkedéssel és funkcionalitással bíró objektumok vezérlik, melyek különféle feladatok végrehajtására szakosodnak. Ellentétben a „mindentudó”, egybázisú programokkal, az OOP-elvet követő programok könnyen bővíthetők, javíthatók és számon tarthatók – viszont az objektumok is szervezhetők rosszul:

* egy objektum **túl sokat** tud: sok, egymástól teljesen független funkciót lát el, mely több objektum között szétosztható lenne 🡪 az átláthatóság csökken, a komplexitás megnő
* egy objektum **túl keveset** tud: az objektum nem lát el elegendő funkciót, mert kis kódrészleteket tartalmaz, egy feladat részleges megvalósítása (függvényszintű kódok)

Mégis, az objektumorientált elv sokkal hatékonyabbnak bizonyul, mint a monolitikus programozás, mivel kevés vezérlő elv betartása és eszköz használata mellett könnyen és viszonylag hibamentesen kivitelezhető.

## Osztály és példány – az objektumok megtestesülése

Rengeteg OOP-re épülő nyelv használ objektumokat eszközül, egy alapszintű eszközkészlet kiterjesztéseként. Webfejlesztésben és üzleti nyelvekben egyaránt megtalálhatók OOP-vonulatok,