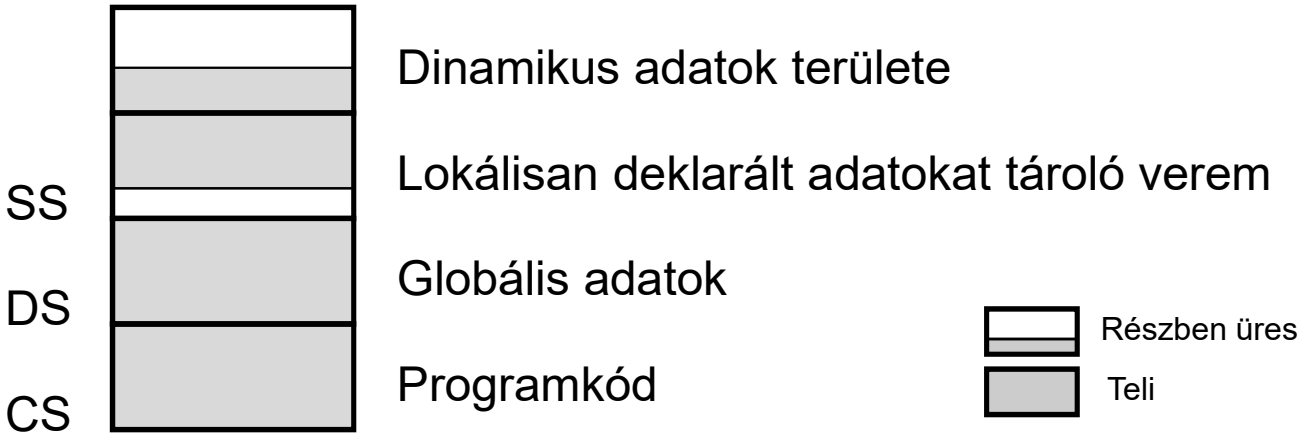
Tárolási osztályok

- *Az azonosító tárolási osztálya*
- *Az azonosító élettartama*
- *Az azonosító érvényességi tartománya*
- *Az azonosító láthatósága*
- *Összefoglaló táblázat a tárolási osztályokról*
- *A program fordítása*
- *A programtesztelés és nyomkövetés eszközei*

Az azonosító tárolási osztálya

A tárolási osztályok a memória szegmentálhatóságával vannak összefüggésben. A számítógép mikroprocesszora rendelkezik négy olyan szegmensregiszterrel, amelyek a memória tetszőleges 64 kByte-os területének az első byte-jára tudnak mutatni. Ezeket a regisztereket CS, DS, SS és ES jelekkel azonosítják. Általában a CS (code segment) regiszter mutat a lefordított programkód elejére, a DS (data segment) mutat a program globális adatait tároló terület elejére, az SS (stack segment) mutat a lokális adatokat tároló veremre és az ES (extra segment) mutathat további adatterületre. A dinamikus memóriafoglalás területe a HEAP (halom).



A memórián kívül még a processzor néhány másik regisztere is szolgálhat változók értékének tárolására.

A megismert tárolási helyek és a négy tárolási osztály a következőképpen párosítható:

globális adatok területe	-	***static***
globális adatok területe	-	***extern***
verem	-	***auto***
regiszter	-	***register***

*A tárolási osztályokat változónevek előtt szerepeltethetjük, a **static** és az **extern** osztálynévvel függvényazonosítókat is elláthatunk, módosítva ezzel a programbeli helyükből eredő alapértelmezett osztálybasorolást, valamint az azonosító élettartamát. Példa az alkalmazás formájára:*

```
register int valt1;
extern float hatvany( float alap, float kitevo);
```

*A globális jellegű **static** és **extern** változók a programban egyszer jönnek létre, ha adtunk nekik kezdeti értéket, akkor azzal, egyébként 0 induló értékkel. Amíg a main() függvénnyel azonos szinten definiált azonosítók alapértelmezetten **extern** osztályúak, azaz másik programmodul-fájlból (ott **extern**-nek újradeklarálva) elérhetőek, addig a **static** osztálymódosítót mindig nekünk kell kiírni.*

*A blokkon belül, lokálisan definiált változók annyiszor jönnek létre a veremben, vagy regiszterben, ahányszor a program ráfut a blokkra, és annyiszor veszik fel újra a megadott kezdőértéküket, annak híján azonban értékük definiálatlan! A változók általában **auto** tárolási osztállyal a veremben jönnek létre, de bizonyos feltételek esetén a regiszter tárolási osztály megadása nélkül is regiszterbe kerülhetnek. A regiszteres változó jelentősen gyorsabb programfutást eredményez, azonban a változó regiszterbe kerülésében a **register** osztály előírásán kívül implementáció-függő jellemzők is közrejátszanak.*

Az azonosító élettartama

A programobjektum tárolási osztálya szoros kapcsolatban van az élettartamával, azaz azzal a programfutáson belüli időtartammal, ameddig az objektum létezik, foglalja a memóriát, vagy egy regisztert.

Statikus (globális, permanens) élettartam:

*Minden függvényazonosító és a függvényekkel azonos (külső) szinten definiált változó **statikus**, azaz a teljes programfutás idején élő és memóriát foglaló objektum. Ebbe természetszerűen beletartoznak a más fájlokban **extern** tárolási osztállyal újradeklarált objektumok is. (A deklarálás nem jár memóriafoglalással, csak utalás egy definiált objektumra.)*

*A **static** tárolási osztályba kényszerített lokális , blokkon belül definiált változók, melyek egyébként **auto** tárolási osztályba (verem), vagy regiszterbe kerültek volna, a globális memóriában jönnek létre és teljes programfutás alatt foglalják a memóriát (de csak a blokkban láthatóak).*

Automatikus (lokális, időszakos) élettartam:

Az osztálybasorolás nélkül, vagy regiszteres osztálybasorolással ***blokkban definiált változók*** *számára arra az időtartamra, amig a program el nem éri a blokk végét, a veremben, vagy a regiszterben foglalódik memória. Hasonlóan a függvényparaméterek számára a függvény meghívásakor lefoglalt memória a függvényblokk végéig él.*

Dinamikus élettartam

A programozó által programfutás közben a dinamikus adatok területén (Heap) lefoglalt (allokált), majd felszabadított változók dinamikus, programozó által diktált élettartammal rendelkeznek.

Az azonosító érvényességi tartománya

Az azonosító érvényességi tartománya az azonosító deklarálásának, ill. definiálásának helye által meghatározott és a hivatkozhatóságnak, elérhetőségnek szükséges feltétele.

***Blokk szintű érvényességi tartomány**a van azon azonosítóknak, amelyeket blokkban deklaráltunk, ill. definiáltunk és a megadástól a blokk végéig tart.*

***Fájl (programmodul) szintű érvényességi tartomány**a van a fájlban a main() függvény szintjén deklarált, ill. definiált változóknak és a fájl függvényazonosítóinak. Az érvényességi tartomány a deklarálástól, ill. definiálástól a fájl végéig tart.*

***Prototípus szintű érvényességi tartomány** vonatkozik a függvényprototípusok paramétereire és a megadástól a paraméterlista végéig tart.*

*A deklarálás mind blokk, mind fájlszintnél történhet **extern** osztálymeghatározóval, ilyenkor az objektum más fájlban került definiálásra.*
Függvény nem definiálható másik függvény blokkjában, de deklarálható.

Az azonosító láthatósága

Egy azonosító akkor hivatkozható, elérhető, ha látható. A láthatósághoz az azonosítónak élnie kell és a hivatkozásnak az azonosító érvényességi tartományában kell történnie.

*Egymásbatokozott blokkokban **ugyanazon nevű változó** újradefiniálható, de ilyenkor a saját automatikus (lokális) élettartama idejére **elfedi**, láthatatlanná, és ezzel elérhetetlenné teszi **a korábban definiált változót**.*

*A következő példához még tudnunk kell, hogy ha egy blokkban definiált (lokális) változót **static** tárolási osztályba kényszerítünk, akkor az globális, az egész programfutás alatt memóriát foglaló lesz, de csak abban a blokkban lesz érvényes, amelyben a definiálás történt. Ha a blokkból kilép, majd újra belép a program végrehajtás közben, a változó a kilépéskori értékével jelenik meg újra.*

*A main() függvény szintjén, globális élettartammal definiált azonosító pedig, amely alapértelmezetten **extern**, azaz külső fájlból is elérthető lenne, a **static** osztályba átsorolva nem lesz továbbra elérhető külső fájlból még abban megadott **extern** deklarálással sem.*

```
#include <stdio.h>
main()                  /* Mintaprogram a láthatóságra  */
{
   int valt;
   valt = 8;
   printf("Nagyblokkban= %d", valt );              /*    8   */
   {
      int valt;                          /* elfedi a korábbi valt-ot   */
      valt = 5;
      printf("\nKisblokkban= %d", valt );          /*    5    */
   }
   printf("\nNagyblokkban= %d", valt );           /*    8    */
}
```

```
#include <stdio.h>
void elj( int k );      /*   prototipus   */
main()                  /*    Mintaprogram a static tárolási osztályra    */
{
   elj(1);              /*  5   */
   elj(2);              /* 10   */
}
void elj( int k )
{
   static int lokbolglob;    /*     static-nak definiáljuk    */
      /*   k > 1 esetén a statikusan megörzött értékkel számol    /*
   if ( k == 1 )  lokbolglob = 5;
   else lokbolglob = k * lokbolglob;
   printf( "\nlokbolglob= %d", lokbolglob );
}
```

Összefoglaló táblázat a tárolási osztályokról

megadás helye	változó/ *fv defin./* prototípus	def/ dekl	tárolási osztály alapért/ kiírt	élet-tar-tam	érvényesség	lát-ha-tó-ság
1. fájlban fájlszin-ten	v1	def	**extern**	glob	F1-ben def→fájlvégig	
	v2	dekl	**extern**	glob	F1-ben dekl→fájlvégig	
	v11	def	**static**	glob	csak F1-ben def→fájlvégig	
	fv1( ){ }	def	**extern**	glob	F1-ben def→fájlvégig	
	fv11();	dekl	**extern**	glob	F1-ben dekl→fájlvégig	
	fv2();	dekl	**extern**	glob	F1-ben dekl→fájlvégig	
1. fájlbeli blokkban	vb1	def	**auto, v. register**	lok	a blokkban a blokkban	
	vb11	def	**static**	glob	a blokkban	
	v2	dekl	**extern**	glob	a blokkban	
	fvb1();	dekl	**extern**	glob	a blokkban	
	fv2();	dekl	**extern**	glob	a blokkban	
2. fájlban fájlszin-ten	mint fent, csak 1⇔2 csere					
2. fájlbeli blokkban	mint fent, csak 1⇔2 csere					

Látható és elérhető, ha élő és érvényes és nem elfedett

A program fordítása

*A fordítás szolgál a futtatható **.exe** kiterjesztésű fájl előállítására. Három fő része az előfeldolgozás, a szorosan vett fordítás és az összeszerkesztés.*

Az előfeldolgozás szövegmanipulációkkal kezeli a megjegyzéseket, a # kezdetű, előfeldolgozónak szóló sorokat, stb.

*A fordító előállítja a szövegfájlból a **.obj** kiterjesztésű tárgy-kódot, amelyben lévő külső objektumhivatkozásoknak megfelelő programrészek beszerkesztése az **.exe** programba a szerkesztő (linker) feladata.*

A programtesztelés és nyomkövetés eszközei

A szintaktikailag hibátlan, lefordított program még sokféle, futásidőben jelentkező hibát tartalmazhat, és a kiakadás nélkül futó program is működhet a programíró szándékaitól eltérő módon. A körülményes, vagy egyáltalán el sem végezhető programhelyesség-bizonyítás helyett megszokott dolog a programok alapos tesztelése. A program működésének futás közbeni nyomonkövetésére a Borland C fejlesztő környezete a következő eszközöket nyújtja:

Trace into F7: *Soronkénti futtatás a függvények blokkjába is belépve*
Step over F8: *Soronkénti futtatás a függvények belsejének átlépésével*
Run to cursor F4: *A program (tovább)futtatása a kurzort tartalmazó sorig*
Inspect Alt+F4: *Változók, függvények memóriabeli elhelyezkedésének vizsgálata*
Evaluate/modify Ctrl+F4: *Változók értékének megtekintése, módosítása*
Add watch Ctrl+F7: *Kiválasztott változók tartalmának állandó kijeleztetése a Watch ablakban*
Toggle breakpoint Ctrl+F8: *Töréspont (ideiglenes programmegállítás) elhelyezése a program valamelyik sorában.*