17 - Programszerkezet

Blokk, hatókör, láthatóság. Automatikus, statikus és dinamikus élettartam, szemétgyűjtés. Konstruktor, destruktor. Objektumok másolása, összehasonlítása. Programegységek, névterek. Alprogramok, paraméterátadás. Túlterhelés.

1. Blokk

A **blokk** olyan programszerkezeti alapegység, amely segítségével a forrásszöveg egy része **egységbe zárható**, továbbá a **hatókör** és a **láthatóság szűkítése** megvalósítható. A blokkok kezdetét és végét külön szimbólum jelzi, ez lehet például `{` és `}` vagy `begin` és `end`.

Blokkok például az elágazások, a ciklusok, a függvények, az eljárások, objektumorientált programozási nyelvekben az osztályok, a metódusok. Számos programozási nyelvben ezeken belül is definiálhatunk blokkokat.

Egyes nyelvekben (például Ada, PL/SQL) a blokkoknak van külön **deklarációs** és **végrehajtási rész**e, míg más nyelvekben (például C/C++, Java) ezek nem különülnek el.

2. Hatókör

Deklarációkor a programozó **összekapcsol** egy **entitást** (például egy változót vagy függvényt) egy **névvel**. A hatókör alatt a forrásszöveg azt a részét értjük, amíg ez az **összekapcsolás érvényben van**. Ez általában annak a blokknak a végéig tart, amely tartalmazza az adott deklarációt.

3. Láthatóság

A láthatóság a **hatókör részhalmaza**, a programszöveg azon része, ahol a **deklarált névhez** a **megadott entitás tartozik**. Mivel az egymásba ágyazott blokkokban egy korábban már bevezetett nevet más entitáshoz kapcsolhatunk, ezért ilyenkor a külső blokkban deklarált entitás a nevével már nem elérhető. Ezt nevezzük a láthatóság **elfedés**ének.
Egyes nyelvekben (például C++) bizonyos esetekben (például osztályszintű adattagok) a külső blokkban deklarált entitáshoz **minősített név**vel hozzá lehet férni ekkor is.

4. Élettartam

A változók élettartama alatt a program végrehajtási idejének azt a szakaszát értjük, amíg a **változó számára lefoglalt tárhely** a **változóé**.

4.1. Automatikus élettartam

A blokkokban deklarált lokális változók automatikus élettartamúak, ami azt jelenti, hogy a **deklarációtól** a **tartalmazó blokk végéig tart**, azaz **egybeesik** a **hatókörrel**. A helyfoglalás számukra a **végrehajtási verem** aktuális **aktivációs rekord**jában történik meg.

4.2. Statikus élettartam

A **globális változók**, illetve egyes nyelvekben a **statikusként deklarált változók** (például C/C++ esetén a `static` kulcsszóval) statikus élettartamúak. Az ilyen változók élettartama a **program teljes végrehajtási idejére** kiterjed, számukra a **helyfoglalás** már a **fordítási időben** megtörténhet.

4.3. Dinamikus élettartam

A dinamikus élettartamú változók esetén a programozó foglal helyet számukra a dinamikus tárterületen (*heap*), és a programozó feladata gondoskodni arról is, hogy ezt a tárterületet később felszabadítsa. Amennyiben utóbbiról megfeledkezik, azt nevezzük memóriaszivárgásnak.
Mint látjuk, a dinamikus élettartam esetén a hatókör semmilyen módon nem kapcsolódik össze az élettartammal, az élettartam szűkebb vagy tágabb is lehet a hatókörnél.

5. Szemétgyűjtő

A szemétgyűjtő másik neve a hulladékgyűjtő, az angol ***Garbage Collector*** név után pedig gyakran csak **GC**-nek rövidítik. Feladata a dinamikus memóriakezeléshez kapcsolódó **tárhelyfelszabadítás automatizálása**, és a felelősség levétele a programozó válláról, így csökkentve a hibalehetőséget.

A szemétgyűjtő figyeli, hogy mely változók kerültek ki a hatókörükből, és azokat felszabadíthatóvá nyilvánítja. A módszer hátránya a **számításigényessé**ge, illetve a **nemdeterminisztikussá**ga. A szemétgyűjtő ugyanis nem szabadítja fel egyből a hatókörükből kikerült változókat, és a felszabadítás sorrendje sem ugyanaz, amilyen sorrendben a változók felszabadíthatóvá váltak.
Azt, hogy a hulladékgyűjtő mikor és mely változót szabadítja fel, egy programozási nyelvenként egyedi, összetett algoritmus határozza meg, amelyben rendszerint szerepet játszik a rendelkezésre álló memória telítettsége, illetve a felszabadításhoz szükséges becsült idő. (Például ha egy objektum rendelkezik destruktorral, akkor általában a GC később szabadítja csak fel.)

Összességében a szemétgyűjtő csak annyit garantál, hogy előbb-utóbb (legkésőbb a program futásának végeztével) minden dinamikusan allokált változót felszabadít.

6. Konstruktor

A konstruktor az **objektumok inicializáló eljárása**, akkor fut le, ha egy osztályból új objektumot példányosítunk. **Alapértelmezett konstruktor** alatt a paraméter nélküli konstruktort értjük, a legtöbb programozási nyelv esetén ezt a fordítóprogram automatikusan generálja üres törzzsel, amennyiben nem lett megadva egy konstruktor sem egy osztályban.

Többek között a konstruktorban szokás gondoskodni arról, hogy az objektum dinamikus élettartamú változói számára tárhelyet foglaljunk.

7. Destruktor

A destruktor a konstruktor ellentétes párja, ez az eljárás az **objektumok felszabadítása**kor fut le. **Meghívása automatikus**an megtörténik, attól függetlenül, hogy az objektum felszabadítása automatikusan történik a hatókör végeztével (lokális objektumok esetén), manuálisan a programozó

által (dinamikus élettartamú objektumok esetén) vagy a szemétgyűjtő által (szintén dinamikus élettartamú objektumok esetén).

A desktruktorban szokás többek között az objektum dinamikus helyfoglalású adattagjait és a lefoglalt erőforrásokat felszabadítani.

8. Objektumok másolása

Az objektumok másolása egy speciális konstruktorral, az úgynevezett **másoló konstruktor**ral (*copy constructor*) történik. Ez paraméterül az adott osztály egy példányát kapja meg, és azt a programozó által megadott működési logika szerint lemásolja az éppen inicializált objektumba.

Több programozási nyelv (például C++) fordítóprogramja automatikusan elkészít egy másoló konstruktort, ha a programozó nem definiál sajátot. Ez az alapértelmezett másoló konstruktor lemásolja a forrásobjektum összes adattagjának értékét, ezt nevezzük **sekély másolat**nak (*shallow copy*). Ha az objektum dinamikus foglalású adattagokat is tartalmaz, akkor azoknak nem az értéke, hanem csak a hivatkozása lesz lemásolva, ami általában nem a kívánt működés. Ez esetben saját másoló konstruktor írása szükséges, ami **mély másolat**ot (*deep copy*) készít.

9. Objektumok összehasonlítása

Objektumok összehasonlítása több féle módon is történhet. Az egyik lehetőség a **referencia alapú** egyezés vizsgálat, ekkor két objektumra hivatkozó változót akkor tekintünk egyenlőnek, ha ugyanarra az objektum entitásra mutatnak, azaz a hivatkozott memóriacím megegyezik.

A másik lehetőség, hogy két objektumot akkor tekintünk egyenlőnek, ha ugyanannak az osztálynak a példányai és adattagjaik – vagy azok egy részhalmazának – értéke megegyezik.

10. Programegységek

Programegységnek nevezzük a program egy olyan szintaktikus egységét, mely a program tagolásának, programrészek újrafelhasználhatóságának eszköze. Programegységek például az alprogramok, a csomagok (Ada), az osztályok (C++), a taszkok (Ada) és a sablonok (Ada, C++). A programegységek definíciója két részből áll: **specifikáció**ból, amely leírja, hogy a programegységet hogyan kell használni, és **törzs**ből, mely a programegység megvalósítását tartalmazza.

A programegység **specifikáció**ja az a része, amely leírja, hogy a programegységet hogyan kell használni. Például egy alprogram specifikációja tartalmazza az alprogram nevét, formális paraméterlistáját, ha függvény, akkor a visszatérési értékét, metódusok esetén a `this` konstansságát, egyes programozási nyelvekben (például Java) a láthatósági és egyéb módosítószavakat és a kiváltható kivételek listáját.

A programegység **törzs**e a megvalósítását tartalmazza. A programegységet használó programrészekből nem látszik. Ilyen az Ada csomagok törzse, vagy a C++ metódusok definíciójának kapcsos zárójelekkel határolt része.

11. Névterek

A névterek az osztálykönyvtárak elválasztására szolgálnak. Használatukkal az összetartozó osztályok egységbe foglalhatóak és a különböző célt szolgáló, de egyező nevű osztályok – esetleg globális változók – közötti névütközés elkerülhető.

A névterek tagjaira kívülről **minősített név**vel lehet hivatkozni. A névtereket **fel** is lehet **oldani**, ez esetben a névtér tartalma minősített nevek mellett közvetlenül is elérhetőek.

12. Alprogramok

Alprogramoknak a **függvény**eket, **eljárás**okat és **művelet**eket nevezzük. Segítségükkel a program feladatonként tagolható, a főprogramból az önálló feladatok kiszervezhetőek.

13. Paraméterátadás

Az alprogramoknak szüksége lehet bemenő adatokra és vissza is adhat értékeket. Az alprogramokat általános írjuk meg, saját változónevekkel, ezek a **formális paraméter**ek. Az alprogram meghívásakor az átadott **aktuális paraméter**ek alapján a formális paraméterek értéket kapnak. Az, hogy a formális paraméterek értéke mi lesz, a **paraméterátadás mód**jától függ.

13.1. Szövegszerű paraméterátadás

A makrókban használatosak mind a mai napig. A makró törzsében a formális paraméter helyére beíródik az aktuális paraméter szövege.

13.2. Név szerinti paraméterátadás

Az aktuális paraméter kifejezést újra és újra kiértékeljük, ahányszor hivatkozás történik a formálisra. A paramétert a törzs kontextusában értékeljük ki, így a formális paraméter különböző előfordulásai mást és mást jelenthetnek az alprogramon belül.
Alkalmazása **archaikus** (például: Algol 60, Simula 67).

13.3. Érték szerinti paraméterátadás

Az egyik legelterjedtebb paraméterátadási mód (például: C, C++, Pascal, Ada), bementi szemantikájú. A formális paraméter az alprogram lokális változója, híváskor a vermen készül egy másolat az aktuális paraméterről, ez lesz a formális. Az alprogram végén a formális paraméter megszűnik.

13.4. Cím szerinti paraméterátadás

A másik legelterjedtebb paraméterátadási mód (például: Pascal, C++, C#, Java), be- és kimeneti szemantikájú. A híváskor az aktuális paraméter címe adódik át, azaz a formális és az aktuális paraméter ugyanazt az objektumot jelentik, egy ***alias*** jön létre.

13.5. Eredmény szerinti paraméterátadás

Kimeneti szemantikájú paraméterátadási mód. A formális paraméter az alprogram lokális változója, az alprogram végén a formális paraméter értéke bemásolódik az aktuálisba. Azonban az alprogram meghívásakor az aktuális értéke nem másolódik be a formálisba. (Használja például az Ada.)

13.6. Érték/Eredmény szerinti paraméterátadás

Az érték és eredmény szerinti paraméterátadás összekombinálása, így egy be- és kimeneti szemantikájú paraméterátadási módot kapunk. (Használja például az Algol-W vagy az Ada.)

13.7. Megosztás szerinti paraméterátadás

Objektumorientált programozási nyelvek (például: CLU, Eiffel, Java) paraméterátadási módja. Lényege, hogy ha a formális paraméter megváltoztatható és az aktuális paraméter egy megváltoztatható változó, akkor cím szerinti paraméterátadás történik, egyébként pedig érték szerinti. A paraméterátadás módját külön megadni nem lehet.

Megváltoztatható (***mutable***) objektum alatt azt értjük, hogy tulajdonságai, ezáltal állapota megváltoztatható.

13.8. Igény szerinti paraméterátadás

Ezt a paraméterátadási módot a **lusta kiértékelésű funkcionális nyelvek** (például: Clean, Haskell, Miranda) alkalmazzák. Az aktuális paramétert nem híváskor értékeli ki, hanem akkor, amikor először szüksége van rá a számításokhoz.

14. Túlterhelés

A túlterhelés (*overloading*) segítségével azonos nevű alprogramokat hozhatunk létre eltérő szignatúrával. A **szignatúra** a legtöbb programozási nyelvben az alprogram nevét és a formális paraméterek számát és típusát jelenti, de egyes nyelvekben (például Ada) a visszatérési érték típusa is beletartozik. A túlterhelés elsődleges felhasználási területe, hogy ugyanazt a tevékenységet különböző paraméterezéssel is elvégezhessük.

A fordító az alprogramhívásból el tudja dönteni, hogy a túlterhelt változatok közül melyiket kell meghívni. Ha egyik sem illeszkedik vagy több is illeszkedik, akkor fordítási hiba lép fel.