Számábrázolás, adatábrázolás

A számítógépek memóriájában a számokat *binárisan* (kettes számrendszerben) ábrázoljuk. A bináris alakban felírt szám egy számjegye 0 vagy 1 lehet. (Ezt hívjuk bit-nek.) Egy számjegy értéke a számon belül elfoglalt pozíciójától, azaz a helyiértékétől függ. A kettes számrendszer helyiértékei a kettő hatványai.

Egy tízes számrendszerbeli szám átírható kettes számrendszerbelibe egy közismert, kettővel való osztogatásos módszerrel. Az osztásoknál keletkezett maradék a soron következő számjegy, az osztás eredménye pedig a következő osztás osztandója. Ezt addig csináljuk, amíg az osztás eredménye 0 nem lesz.
Példa: $183_{10} = 10110111_2 = 1*2^7 + 0*2^6 + 1*2^5 + 1*2^4 + 0*2^3 + 1*2^2 + 1*2^1 + 1*2^0$

Egy törtszám bináris alakjának számegyeit helyiérték szerinti csökkenő sorrendben egy kettővel való szorzásos módszerrel lehet előállítani. Az aktuális szorzat egész része a soron következő számjegy, törtrésze a következő szorzás szorzandója. A módszer addig tart, amíg a szorzandó nem 0, de előfordulhat, hogy ez nem következik be, mert a kettes számrendszerbeli alak egy végtelen kettedes tört lesz.
Példa: $0.8125_{10} = 0.1101_2 = 1*(1/2^1) + 1*(1/2^2) + 0*(1/2^3) + 1*(1/2^4)$

Alaptípusok ábrázolása

Az esetek többségében alaptípusok alatt a számtípusokat (egész számok, valós számok), a logikai típust és a karakter típust értjük.

Természetes számok ábrázolása

A számítógépeken 8 bit (1 byte) többszörösein történik az ábrázolás – azért, mert a byte a legkisebb címezhető egység a memóriában. A természetes számok ábrázolásánál egyszerűen kettes számrendszerbe váltjuk a számot. Ha nem áll rendelkezésre elegendő bit, akkor túlcsordulásról beszélünk. Ilyenkor az eredeti szám értéke hibásan rögzül.

Egész számok ábrázolása

Ebben az esetben már az adott szám előjelét is ábrázolni kell az abszolútértéke mellett. Az abszolútértéket a bináris alakjában tároljuk, az előjelet pedig 1 biten. Többféle ábrázolási módszer is kialakult, ezeket röviden leírom:

- Egyenes kód: A számokat s~~+1~~ biten írjuk le úgy, hogy az 1. biten a szám előjelét, a maradék s biten pedig a szám abszolút értékének bináris alakját ábrázoljuk. Az ábrázolható értékek tehát a $[-2^{s-1}+1,2^{s-1}-1]$ intervallumban levő számok (ellenkező esetben túlcsordulás lép fel). Az ábrázolás hátránya, hogy a nullát két bináris kód is jelenti (00000000 és 10000000), továbbá az, hogy két szám összegét a számok előjelétől függő módszerekkel lehet kiszámolni.
- Egyes komplemens (vagy inverz) kód: A számokat s biten ábrázoljuk úgy, hogy abszolút értékének bináris alakja 0-val kezdődjön. Az ábrázolható értékek a $[-2^{s-1}+1,2^{s-1}-1]$ intervallumban levő számok (ellenkező esetben túlcsordulás lép fel). A pozitív egész számokat ezzel a kóddal ábrázoljuk. A negatív számokat pedig úgy, hogy vesszük az abszolút érték bináris alakjának komplementerét (ekkor a szám 1-essel fog kezdődni annak jelzéseként, hogy a szám negatív). Az ábrázolásnak az a hátránya itt is megvan, hogy a nullát két bináris kód is jelenti (00000000 és 11111111), továbbá az, hogy két

szám összeadásakor a tulcsordulási bitet hozzá kell adni az eredményhez, hogy megkapjuk a helyes eredményt (feltéve, hogy az eredmény ábrázolható *s* biten).

- Kettes komplemens kód: Az egész számok általánosan elterjedt kódolása. A módszer nagyon hasonló az egyes komplemens kódhoz: a pozitív számokat ugyanúgy kódoljuk, a negatív számokhoz pedig itt is vesszük az abszolút érték bináris alakjának komplementerét – majd ehhez hozzáadunk 1-et. A 0 bit-tel kezdődő bináris kód itt is azt jelzi, hogy a szám pozitív, míg az 1-gyel kezdődő azt, hogy negatív.

  Példa: -12 kettes komplemens alakja 4 byte-on
  1. Először vesszük 12 kettes komplemens alakját: 12_{10} = 00000000 00000000 00000000 00001100_2
  2. Vegyük a bináris alak komplementerét: 11111111 11111111 11111111 11110011
  3. Adjunk hozzá binárisan 1-et, így megkapjuk a végeredményt:
     -12_{10} = 11111111 11111111 11111111 11110100_2

- Többletes kód: A számokhoz, mielőtt binárisan ábrázoljuk őket, hozzáadunk egy rögzített értéket (eltolás). Az eltolás értékét úgy választjuk meg, hogy az ábrázolni kívánt számok az eltolás után ne legyenek negatívak. Például ha s biten akarjuk a számokat ábrázolni, akkor az eltolás lehet a 2^{s-1}. Ennek bináris alakja (100...00) jelenti a nullát és az ábrázolható számok -2^{s-1} és $2^{s-1}-1$ közé esnek. (Az első számjegy most is a szám előjelét jelzi, de itt az 1 jelenti azt, hogy a szám pozitív, a 0 pedig, hogy negatív, ellentétben az eddigiekkel.) Elsősorban számok nagyság szerinti összehasonlításához, számok összegének és különbségének kiszámolásához alkalmas ábrázolás.

Valós számok ábrázolása

- Fixpontos számábrázolás: Rögzített számú biten lehet ábrázolni külön a szám egészrészét és külön a törtrészét. Az előjel ábrázolására alkalmazhatók az egész számoknál említett módszerek.
- Lebegőbpontos számábrázolás (IEEE 754 szabvány): A számokat $((-2)^*b+1)^*m^*2^k$ alakban írjuk fel.
  - *b*: előjel (0 – pozitív, 1 – negatív)
  - *m*: mantissza ($1 <= m < 2$)
  - *k*: karakterisztika

  Az ábrázolás lehet 4 vagy 8 byte-os. Maga a kód a valós szám előjelbitjéből, utána *s* biten (8, ill. 11) a karakterisztika $2^{s-1}-1$ eltolású többletes kódjából (ekkor $-2^{s-1}+1 <= k <= 2^{s-1}$), végül a mantissza bináris alakjának törtrészéből (23, illetve 52 biten) áll.

  Példa: A -12.25 valós szám lebegőpontos kódja 1 + 11 + 52 biten:
  - Negatív szám, ezért az előjelbit 0 lesz: $b = 0$.
  - A szám abszolút értékének bináris alakja: $12.25_{10} = 1100.01_2$
  - Normalizáljuk az így kapott számot 1 és 2 közé: $1100.01_2 = 1.10001_2 * 2^3$
  - Ábrázoljuk a karakterisztikát többletes kódban 11 biten: $3 + 2^{10} - 1_{10} = 10000000010_2$
  - Állítsuk össze a kódot: 1 10000000010 10001000000000000000 … 00000000

Karakterek ábrázolása

Egy karakterkészlet elemeit, megszámozzuk 0-tól kezdődően és az így kapott karakterkódok bináris számmá átalakítva reprezentálják a karaktereket. Az egyik legegyszerűbb, de igen elterjedt ilyen készlet az ASCII

karakterkészlet, amelyik 256 különböző karaktert tartalmaz – így a sorszámok kódja (0 – 255) 1 byte-on ábrázolható.
Az UTF-8 változó hosszú kódolással ábrázolja a karaktereket – magába foglalja az ASCII karaktereket, ezek kódjai 1 byte hosszúak, de például az ékezetes betűk kódjai már 2 byte-on tárolódnak.

Logikai érték ábrázolása

Kétféle logikai érték van: *igaz* és *hamis*. Ennek megfelelően az ábrázoláshoz 1 bit is elég lenne. Azonban a memória legkisebb címezhető egysége a byte, ezért 1 byte-on ábrázoljuk a logikai értékeket (is): a csupa 0 bit-ből álló byte jelenti a *hamis* értéket, minden egyéb az *igaz* értéket.

Típuskonstrukciók

A típuskonstrukció azt jelenti, hogy már meglevő típusokat használunk fel új típus létrehozására.
A típuskonstrukciók értékhalmazának struktúrája alapján 3 fő típuskonstrukciós módszert különböztetünk meg:

- *Direktszorzat*
- *Unió*
- *Sokaság*/*Iterált*

A típuskonstrukciók műveleteinek két fontos csoportja:

- Típuskonstrukciós műveletek
  - Elem felvétele (adott helyre, elejére, végére, …)
  - Elem törlése (adott helyről, elejéről, …)
- Szelekciós/Kiválasztó műveletek
  - Elem keresése (hely alapján, érték alapján, ...)

Direktszorzat típuskonstrukció

$$T = T_1 \times T_2 \times \ldots \times T_n$$

A típus minden eleme a T_i halmazokból vett elemek direktszorzata. A típus ábrázolása annyi biten történik, ahány biten az egyes résztípusok összesen ábrázolhatók. Fontos megjegyezni, hogy ezen típusok esetén nincs beszúró vagy törlő művelet, hiszen meghatározott számú eleme van a direktszorzatnak (pontosan *n*).
Példa: rekord típusok (pl. Pascal nyelvben), tuple (Haskell)

Unió típuskonstrukció

$$T = T_1 \cup T_2 \cup \ldots \cup T_n$$

A típus egy eleme értékeit n db különböző típus valamelyikéből veszi. Fontos különbség tehát a direktszorzathoz képest, hogy egy unió típuskonstrukció segítségével létrehozott objektumnak (értéknek) nem kell *n* eleműnek lennie. Ez a fajta típuskonstrukció ritkábban fordul elő, mint a másik kettő, van olyan programozási nyelv, amelyben nincs is eszköz a megvalósítására (pl. Python). Sokszor az unió helyett öröklődést használunk.
Legjobban elmagyarázni, hogy miről is van szó, talán egy példán keresztül lehet: tegyük fel, hogy emberekről (férfiakról és nőkről) szeretnénk adatot tárolni. Azonban amikor felvisszük az adatokat, nem

pontosan ugyanazokat az adatokat szeretnénk látni egy férfi és egy nő esetében (pl. a férfiakhoz nem kell leánykori név). Ennek a problémának feloldására alkalmazhatunk unió típuskonstrukciót.
Példa: **C nyelvben az unió megadásának módja:**

```
typedef union { FerfiTipus ferfi; NoTipus no; } EmberTipus;
```

Iterált típuskonstrukció

$$T = T_1^*$$

A típus minden eleme véges számú, azonos típusú elemből áll. Igen gyakran használt konstrukció. A különböző adatszerkezeteket (pl. gráf, fa, verem, ...) ilyen konstrukcióval állítjuk elő. Az ábrázolás módja szerint lehet csoportosítani az iteráltakat.

Szekvenciális ábrázolás

Az adatszerkezet elemei közvetlenül egymás után helyezkednek el a memóriában. Az első elem címe egyben a tömb címe is, a tömb elemeire pedig relatív címzéssel (indexeléssel) férhetünk hozzá.
Példa: Egy négy egész számot tartalmazó tömb esetében a memória egymás utáni 16 byte-ján fog elhelyezkedni a négy szám kettes komplemens kódja. pl. int v[] = {1,0,-5,3} tömb esetén v[1] a 0 elemet fogja jelenteni (a tömb kezdőcímén levő elem mellett 1-gyel helyezkedik el).

A többdimenziós tömb felfogható eggyel kisebb dimenziójú tömbök tömbjeként. Például a mátrix (kétdimenziós tömb) felfogható úgy, mint a sorainak (amelyek egydimenziós tömbök) tömbje (sorfolytonos szemlélet). Ugyanígy felfogható az oszlopainak tömbjeként is (oszlopfolytonos). A memórián belül, példaként a sorfolytonos szemléletet tekintve, a mátrixot sorait a memóriában közvetlenül egymás után tároljuk (ahogy az egydimenziós tömbökön belül a tömbelemeket).

Előnyök: elemek elérése, keresése egyszerű
Hátrányok: törlés és beszúrás nehéz, sok mozgatással jár (törlésnél a keletkező „lukak” eltüntetése, beszúrásnál az új elemnek „helyet kell csinálni” elemek mozgatásával), az elemek száma rögzített

Megjegyzés: A *string*, vagyis a karakterlánc gyakran (pl. C nyelv) egyszerű karaktertömbként van reprezentálva. Ekkor a karakterek kódját sorban egymás után tároljuk el (mint egy *int* tömbben, csak ez most *char* tömb lesz). Vagy egy külön számláló jelzi a *string* hosszát (ez többnyire a karakterlánc kezdetének címén található egész szám), vagy a lánc végén egy speciális, erre a célra fenntartott karakter ('\0') jelzi a végét.

Láncolt ábrázolás

Az elemek fizikai és logikai sorrendje eltérő. A logikai sorrend a soron következő elem helyének tárolásával valósul meg.

- *Statikusan láncolt* ábrázolás: Az *i*-edik elem után az *i*+1-edik elem sorszámát (indexét) tároljuk.
- *Dinamikusan láncolt* ábrázolás: Az *i*-edik elem után az *i*+1-edik elem memóriacímét (referenciáját) tároljuk.

Előnyök: könnyű beszúrás, törlés (csak a soron következő elem lesz más, nem szükséges mozgatás), elemek száma dinamikusan változhat
Hátrányok: nehéz keresés (nem lehet indexelni)

Operátorok, kifejezések

Operandusok: Változók, konstansok, függvény- és eljáráshívások.

Operátorok: Műveleti jelek, amelyek összekapcsolják egy kifejezésben az operandusokat és valamilyen műveletet jelölnek.

Kifejezés: operátorok és operandusok sorozata

Precedencia: A műveletek kiértékelési sorrendjét határozza meg.

Asszociativitás iránya: Az azonos precedenciájú operátorokat tartalmazó kifejezésekben a kiértékelés iránya.

Az operátorokat háromféleképpen írhatjuk az operandusokhoz képest:

- Infix: Egy operátort a két operandusa közé kell írni (tehát csak kétoperandusú műveletek operátorait lehet így írni).
  Amikor egy kifejezésben több operátor is szerepel, akkor a különböző operátorok végrehajtási sorrendjét az operátorok precedenciája dönti el. Amelyik operátor precedenciája magasabb (pl. a szorzásé magasabb, mint az összeadásé), az általa jelölt műveletet értékeljük ki először. Ugyanazon operátorok végrehajtási sorrendjét pedig az asszociativitás iránya dönti el (pl. a balasszociatív azt jelenti, hogy balról jobbra haladva kell végrehajtani). Ezeket a (programozási nyelvekbe beépített) szabályokat zárójelek segítségével lehet felülírni.
  Példa: A * (B + C) / D
- Postfix: Az operátorokat az operandusaik mögé írjuk. A kiértékelés sorrendje mindig balról jobbra történik – tehát egy *n* operandusú operátor a tőle balra levő első *n* operandusra érvényes.
  Példa: A B C + * D /
  Ugyanez zárójelezve (felesleges): ((A (B C +) *) D /)
- Prefix (Lengyelforma): Az operátorokat az operandusuk elé írjuk. A kiértékelés sorrendje balról jobbra történik.
  Példa: / * A + B C D
  Ugyanez zárójelezve (felesleges): (/ (* A (+ B C)) D)
  Habár a prefix operátorok esetén is balról jobbra történik a kiértékelés, viszont ha egy operátortól jobbra egy másik operátor következik, akkor értelemszerűen az ehhez az operátorhoz tartozó műveletet kell először végrehajtani, hogy a bal oldalit is végre tudjuk hajtani. A fenti példában is a szorzást az osztás előtt, az összeadást pedig a szorzás előtt kell elvégezni.

Fordított lengyelforma

Megjegyzések:

- A fenti példák mindegyikénél ugyanazt a kifejezést írtam fel, csak infix, postfix és prefix módon. A kifejezés helyes kiértékelési sorrendje: összeadás, szorzás és végül az osztás.
- Természetesen lehet „vegyesen” is írni az operátorokat. Például: a kétoperandusú operátorokat (ld. „+”, „-”, „*”, …) a gyakorlatban infix jelölésekkel szoktuk írni. Azonban az ezeket a műveleteket tartalmazó kifejezésekben előkerülhet például a hatványozás, vagy a gyökvonás művelet, amelyeket nem lehet infix módon jelölni.

Logikai operátorokat tartalmazó kifejezések kiértékelése

Az ilyen kifejezéseknek kétféle kiértékelése létezik:

- Lusta kiértékelés: Ha az első argumentumból meghatározható a kifejezés értéke, akkor a másodikat már nem értékeli ki.
- Mohó kiértékelés: Mindenféleképpen megállapítja mindkét argumentum logikai értékét.

A két kiértékelési módszer bizonyos esetekben különböző eredményt adhat:

- A 2. argumentum nem mindig értelmes
  Példa: `if ((i>0) && (T[i]>=10)) then ...`
- A 2. argumentumnak van valamilyen mellékhatása.
  Példa: `if ((i>0) || (++j>0)) then T[j] := 100`

Utasítások, vezérlési szerkezetek

Egyszerű utasítások

- Értékadás: Az értékadás bal oldalán egy változó, a jobb oldalán bármilyen kifejezés állhat. Az értékadással a változóhoz rendeljük a jobb oldali kifejezést. Figyelni kell arra, hogy a bal oldali változó típusának megfelelő kifejezés álljon a jobb oldalon (vagy létezik implicit konverzió, pl. C++-ban az egész és logikai típus között).
- Üres utasítás: Nem mindenhol lehet ilyet írni. A lényege, hogy nem csinál semmit. Ezzel le lehet kódolni például Adá-ban egy üres `begin-end`-et, vagy üres ciklust.
  Megjegyzés: A legtöbb fordító az üres, vagy értelmetlen ciklusokat (amikor a ciklusbeli utasítások eredményét nem használjuk fel sehol) kioptimalizálja (nem hajtódik végre futás közben).
- Alprogramhívás: Alprogramokat nevük és paramétereik megadásával hívhatunk.
  Példák: `writeln(„Hello”);`
  `int x = sum(3,4);`
- Visszatérés utasítás: Az utasítás hatására az alprogram végrehajtása befejeződik. Ha az alprogram egy függvény, akkor meg kell adni a visszatérési értéket is.
  Példa: `return x+y;`

Vezérlési szerkezetek

Utasításblokk: Nem vezérlési szerkezet, de vezérlési szerkezeteknél gyakran használatos nyelvi eszköz. A blokkon belüli utasítások „összetartoznak”. Ez több esetben is jól alkalmazható nyelvi elem:

- Vezérlési szerkezetekben: Az adott vezérlési szerkezetekhez tartozó utasításokat különíthetjük el.
- Az olyan nyelvekben, amelyekben deklaráció csak a program elején található deklarációs blokkokban lehetséges (pl. Ada), van lehetőség arra, hogy a programkód későbbi részében nyissunk egy blokkot, ahol deklarációk is szerepelhetnek.
- Elágazás: Az elágazás egy olyan vezérlési szerkezet, amellyel meghatározhatjuk, hogy bizonyos (blokkban megadott) utasítások csak a megadott feltétellel jöhessenek létre. Több feltételt is megadhatunk egymás után (`if` L1 `then` … `else if` L2 `then` … `else if` L3 `then` …). Megadhatjuk azt is, hogy mi történjen, ha egyik feltétel sem teljesül (`if` L `then` … `else` …).

Elágazásokat lehet egymásba ágyazni (`if … then if …`)
„*Csellengő else*” („*Dangling else*”) *probléma*: Azokban a nyelvekben lép fel, ahol egy feltétel egy utasításblokkját nem zárja le külön kódszó (pl. `endif`). Ekkor abban az esetben, ha elágazásokat egymásba ágyazunk, a következő probléma léphet fel:

Példa: `if (A>B) then if (C>D) then E:=100 else F:=100`

A fenti esetben nem lehet megállapítani, hogy a programozó az else kulcsszót melyik elágazásra értette.

- Ciklus: Egy utasításblokk (*ciklusmag*) valahányszori végrehajtását jelenti.
  - Feltétel nélküli ciklus: Végtelen ciklust kódol, kilépni belőle a strukturálatlan utasításokkal lehet (ld. lentebb), vagy `return`-nel (, esetleg hiba fellépése esetén).
    Példák: `while (true) …` //speciális elöltesztelő ciklus
    `loop … endloop`
  - Elöltesztelő ciklus: A ciklus a ciklusmag minden végrehajtása előtt megvizsgálja, hogy az adott feltétel teljesül-e. Ha teljesül, akkor végrehajtja a magot, majd újra ellenőriz. Különben a ciklus után folytatódik a futás.
    Példa: `while (i>1) do i := i/2;`
  - Számlálásos ciklus: Ebben a vezérlési szerkezetben megadhatjuk, hogy a ciklusmag hányszor hajtódjon végre.
    Példa: `for I in 1..10 loop kiír(„Hello”); end loop;`
    A számlálás úgy történik, hogy egy változóban tároljuk, hogy „hol tartunk” - ezt hasonlítjuk össze minden ciklus elején a kifejezéssel, amit meg kell haladnia a változónak (`I`). Tehát felfogható egy elöltesztelő ciklusként is, ahol a ciklusfeltétel az „ `i<10`” és a ciklusmag végén növelni kell `i`-t.
  - Hátultesztelő ciklus: Az a különbség az elöltesztelőhöz képest, hogy a feltételt a ciklusmag végrehajtása után ellenőrizzük – tehát itt a ciklusmag 1-szer mindenképpen lefut.
    Példa: `do c = getchar(f) while c!=”Q”;`
  - foreach: Akkor használatos, ha egy adatszerkezet minden elemére végre akarjuk hajtani a magot. Tulajdonképpen ez is egy elöltesztelő ciklus (a ciklusfeltétel az, hogy a végére értünk-e az adatszerkezetnek).
    Példa: `foreach (int v in Vect) ++v;`
    Megjegyzés: Nem minden programozási nyelvben a `foreach` a kulcsszó ehhez a ciklushoz.
    Nem minden programozási nyelvben van foreach szerkezet. (Pl. C/C++)

Strukturálatlan utasítások

- Ciklus megszakítása: A ciklusból való „kiugrásra” (tehát annak azonnali befejezésére) használható. Gyakran végtelen ciklus megszakítására használjuk, vagy hátultesztelő ciklus kódolására (ahol nincs erre beépített vezérlési szerkezet, például Ada).
- Goto utasítás: Használata: A programkódban címkéket definiálhatunk, majd a goto utasítással egy ilyen címkéhez irányíthatjuk a vezérlést. Vezérlési szerkezeteket is lehet vele kódolni. Túlzott használata olvashatatlan kódhoz vezethet.

Rekurzió

Rekurzív alprogram: Olyan alprogram, amelynek kódjában szerepel önmagának a meghívása.
Mindenképpen kell, hogy legyen a rekurziónak *megállási feltétele* – tehát egy olyan feltétel (egy elágazással együtt), amelynek teljesülése esetén nem történik rekurzió, így az összes, folyamatban levő rekurzív hívás végre tud hajtódni (ellenkező esetben végtelen ciklusba jutunk!). Ebből a szempontból tehát a rekurzió hasonlít a ciklusra (ciklusfeltétel = megállási feltétel).
Tipikus ***példa*** a faktoriális program (Kozsik Tamás diáiból, Ada nyelven):

```
function Faktoriális ( N: Natural ) return Positive is
begin
          if N > 1 then return N * Faktoriális(N-1);
          else return 1;
          end if;
end Faktoriális;
```

Kivételkezelés

Kivétel: Olyan esemény, ami eltér a várttól, a megszokottól, a gyakoritól. Jelezhet

- *hibát*: Sokféle hiba léphet fel, pl.: tömb túlindexelése, null pointer-re hivatkozás, hálózati kapcsolat megszakadása, …
- *speciális eseményt, esetet* (amely nem gyakran történik, esetleg nem olyan fontos)

A mai nyelvekben már a kivétel egy külön típus és a kivételkezelés nyelvileg támogatott. Az olyan nyelvek esetén, ahol ilyen nincs (pl. Pascal), a különböző visszatérési értékeket szokták kivételként használni (– a kivételkezelés pedig egy visszatérési érték szerinti elágazásként valósul meg). A másik lehetőség, hogy nem foglalkozunk a kivételekkel – ez esetenként megéri a jobb olvashatóság, átláthatóság miatt.

Kivétel kezelése: A program egy utasításának végrehajtása közben léphet fel egy kivétel. Ekkor két dolgot tehetünk:

1. Engedjük a kivételt *terjedni*. Egy kivétel mindig a hívási verem mentén terjed – tehát egy alprogramból az őt meghívó alprogramba (esetleg a főprogramba) vagy a tartalmazó blokkba és így tovább. A terjedés azt jelenti, hogy ahova terjed, ott is fellép a kivétel (mindig az alrpogramhívás/blokktartalmazás helyén). Ha egy kivétel fellép és nem kezeljük le, akkor az adott alprogram végrehajtása megszakad. Ha a főprogramban fellép egy kivétel és nincs lekezelve, akkor a program futása megszakad és valamilyen információt kapunk a fellépő hibáról (hol lépett fel, mi a hiba megnevezése, ...).
2. *Lekezeljük* a kivételt: Egy kivétel lekezelése megállítja annak terjedését. A kivételek lekezelésekor megmondhatjuk, mi történjen. Például:
   - Próbáljuk folytatni a működést a kivétel ellenére.
   - Hárítsuk el a hibát és próbálkozzunk újra.
   - Valamit még csináljunk a kilépés előtt (pl. adatbázis zárolása).

A kivétel terjedési útját ismerve megválaszhatjuk, hogy hol kezeljük le az adott kivételt. Nem mindig érdemes azonnal lekezelni egy kivételt (pl. csak az alprogramot meghívó alprogram fér hozzá egy bizonyos változóhoz).

Bizonyos nyelvek (pl. Java) támogatnak egy olyan opciót is a kivételkezelés esetén (`finally`), amely segítségével definiálhatunk egy olyan utasításblokkot, melynek utasításai minden esetben végrehajtódnak (akár fellépett kivétel, akár nem, akár lekezeltük a kivételt, akár nem). Ez fontos lehet pl. abban az esetben, ha egy file-t mindenképpen szeretnénk lezárni egy műveletsorozat végén (amelyben kivétel léphet fel).

Kivételeket természetesen mi magunk is kiválthatunk a saját programjainkban. Akár arra is lehetőség van, hogy egy kivételt kezelő ágban az adott kivételt újra kiváltsuk, mert nem tudtuk a kivételkezelést az adott helyen még befejezni.
Amely nyelvekben a kivétel külön típust jelöl, ott lehetőség van saját kivétel deifinálására is. Ezeket természetesen később ugyanúgy kiválthatjuk és lekezelhetjük.

Adatabsztrakció

Az adatabsztrakció azt jelenti, hogy valamilyen módszerrel a feladat szempontjából lényegtelen információt elhagyjuk/elrejtjük. Az adatabsztrakció használata fontos és elvárt. Előnyei:

- *egységbe zárás (encapsulation)*: A program funkció szempontjából egybetartozó részei a kódban is egysége lesznek zárva, külön a nem odatartozó részektől. Ezáltal a kód értelmezése könnyebb, maga a kód pedig átláthatóbb lesz.
- *információ elrejtés*: Egy adott P programegység bizonyos adattagjai el vannak rejtve az egységet használó külső program elől. Csak a kívülről látható (használható) dolgokat feltételezhetjük P-ről. Ez azért előny, mert így a P-t használó programban nem kell ezekre az elrejtett részletekre figyelni.
- *modularizáció*: Ha programunkat modulokra (különálló programegységekre, akár a különálló részeket külön file-okra) bontjuk, a program megírása is könnyebb, gyorsabb.
- A modulokra bontott program *karbantartása* is könnyebb: egyrészt a jobb olvashatóság, másrészt a robosztusság (kód újrafelhasználása) miatt.

Osztályok

Osztály: Az *objektumorientált programozás* alapja, hogy az „összetartozó” adatokat (egy adott feladat megoldása szempontjából) műveleteket *egységbe zárjuk*.
Ha egy osztályt használni akarunk, akkor legtöbbször (kivéve pl. a statikus metódusokat, de ez most nem olyan fontos) *példányosítani* kell az osztályokat. Egy osztály példányosításakor egy, az adott osztálybeli *objektum* jön létre. Minden példányosított objektum osztálya egyértelműen meghatározott. Tehát egy osztályt tekinthetünk „hasonló” objektumok gyűjteményének. A hasonlóság itt reprezentációs, illetve viselkedésbeli hasonlóságként értendő.
A programozásban az elméleti típusokat osztályokkal reprezentáljuk.

Öröklődés

Öröklődés: Egy osztály kiegészítése új tagokkal (változókkal, metódusokkal). Az eredeti osztály lesz a *szülő* osztály, a szülő kiegészítésével kapott osztály pedig a *gyerek* osztály. Egy szülőnek több gyereke is lehet,

míg egy gyereknek bizonyos nyelvekben (pl. Java) csak egy szülője.
Egy gyerek osztály „örökli" a szülője *public* és *protected* elérésű adattagjait (azaz rendelkezik velük).
Megjegyzés: (adattagok elérhetőségének definiálása)

public: Kívülről elérhető és örökített adattagok.
protected: Kívülről nem elérhető, de örökölt adattagok.
private: Kívülről nem elérhető és nem örökített adattagok.

Az öröklődés egy gyakran használatos programozási eszköz. Főbb előnyei:

- Átláthatóság, olvashatóság: Az osztályok közötti kapcsolatot egyértelműen és könnyen érthetően jelölhetjük, ha megvan köztük az öröklődési kapcsolat.
- Kód újrafelhasználás (robosztusság): Abban az esetben, ha több osztály tartalmazna ugyanolyan funkciót betöltő változó(ka)t, vagy metódus(oka)t, akkor ezeket a közös tagokat egy közös ősosztályba tenni – ekkor ezeket a tagokat csak egyszer kell kódolni.
- Karbantarthatóság: Ha például valamelyik ősosztályban definiált metódust kell átírni, vagy oda egy új metódust bevezetni, könnyebb a dolgunk, hiszen csak egy helyen kell változtatni.

Osztályhierarchia: Az öröklődési relációt gráfként megadva kapjuk az osztályhierarchiát. Ha csak egyszeres öröklődés van megengedve, ott ez egy irányított erdőt jelent.

Altípusosság

Mint azt fentebb írtam, a gyermek típus rendelkezik a szülőjének összes attribútumával. Tehát minden eseményre reagálni tud, amire a szülő is. A gyermek minden olyah helyzetben használható, amiben a szülő is. Erre mondhatjuk azt, hogy a gyermek típusa a szülő típusának egy altípusa.

- Ugyanaz a művelet meghívható a szülő és a gyermek osztállyal is, tehát több típussal rendelkezik a művelet.
- Egy rögzített típusú változó hivatkozhat több különböző típusú objektumra (szülő típusú referenciának értékül adható egy leszármazott típusú példány).
- Egy metódus aktuális paraméterének típusa lehet a megadott formális paraméter leszármazottja.

Ez a fajta többalakúság az *altípusos polimorfizmus*.

Statikus kötés, dinamikus kötés

statikus típus: A változó deklarációjában megadott típus. Fordítás során egyértelműen eldől, nem változhat futás során. A statikus típus határozza meg, hogy mit szabad csinálni az objektummal (pl. hogy milyen műveletek hívhatók meg rá).

dinamikus típus: A változó által hivatkozott objektum típusa. Vagy a statikus típus leszármazottja, vagy maga a statikus típus. Futás során változhat.

Példa: `Object o = new String(„Hello");` → statikus típus: `Object`
→ dinamikus típus: `String`

statikus kötés: A változó statikus típusa szerinti adattagokra lehet hivatkozni.

dinamikus kötés: A változó dinamikus típusa szerinti adattagok használhatók.

A kötés akkor fontos, ha a hívott művelet, vagy a hivatkozott változó a statikus és a dinamikus típusban különbözik, vagy esetleg a statikus típusban nem is létezik (ekkor a dinamikus típusban sem hivatkozhatunk az adattagra).

Példa:
```
Alkalmazott a = new Alkalmazott(...);
Főnök f = new Főnök(...);
int i = a.pótlék();
int j = f.pótlék();
a = f;
int k = a.pótlék();
```

Többféleképpen lehet a kötéseket meghatározni a különböző nyelvekben:

- Java: Minden esetben dinamikus kötés van (az örökölt metódusok törzsében is).
- C++: A művelet definiálásakot lehet jelezni, ha dinamikus kötést szeretnénk (`virtual`).
- Ada: A híváskor lehet jelezni, ha dinamikus kötést szeretnénk.

Generikus programozás (Generic programming)

Generikus programozás: Algoritmusok, adatszerkezetek általánosított, több típusra is működő leprogramozása (pl. generikus rendezés, generikus verem, …). Ezt az általános kódot nevezzük sablonnak (template).

Bizonyos nyelvekben (Ada, C++) egy sablont példányosítani kell – ekkor a kívánt típusokat, objektumokat (a sablon definíciónak megfelelően) a sablonnak paraméterként megadva példányosul a szóban forgó sablon. (Ugyanúgy, mint pl. amikor egy függvénynek megadjuk az aktuális paraméterét.) Ezt nevezzük generatív programozásnak: a program a megadott sablon alapján létrehoz egy „igazi” programegységet (program generál programot).
Példa: Egy verem sablon példányosításakor megadjuk, hogy milyen típusú elemeket tároljon a verem (típus paraméter) és hogy hány elem fér a verembe (objektum paraméter).
Egy sablon paramétere alprogram is lehet (C++, Ada, Java).
Példa: Egy rendezésnek megadjuk, hogy milyen művelet alapján rendezzen.
Természetesen lehet alapértelmezett sablonparaméter is.

Egy sablon definiálása esetén természetesen a sablont nem lehet minden típusra használni. Egy sablon attól lesz sablon, hogy több típusra is működik. Azonban megszorításokat tehetünk (és tennünk is kell) arra, hogy milyen típusokra lehessen azt használni, hogyan lehessen a sablont példányosítani.
Jó példa erre az Ada nyelv, ahol a sablon specifikációja egy „szerződés” a sablon törzse és a példányosítás között:

- A sablon törzse nem használhat mást, csak amit a sablon specifikációja megenged neki. (A törzset nem feltétlenül kell, hogy ismerjük példányosításkor.)
  Példa:
  ```
  with function "<" (A, B: T) return Boolean is <>;
  function Maximum ( A, B: T ) return T;
  function Maximum ( A, B: T ) return T is
   begin
    if A < B then return B; else return A; end if;
   end Maximum;
  ```

- A példányosításnak biztosítania kell mindent, amit a sablon specifikációja megkövetel tőle.
  Példa:
  ```
  function I_Max is new Maximum( Integer, "<" );
  function I_Min is new Maximum( Integer, ">" );
  function F_Max is nem Maximum( Float );
  ```

Viszont például a C++-ban a sablonszerződés nem így működik. Ott a sablon specifikációja az egész definíció. Példányosításkor ezt is ismerni kell, hogy tudjuk, hogyan példányosíthatunk. Az információelrejtés elve tehát sérül.
Más nyelvekben (pl. Java, funkcionális nyelvek) nem kell a sablonokat példányosítani – mindig ugyanaz a megírt kód hajtódik végre, csak épp az aktuális paraméterekkel.

Források:

- http://nik.uni-obuda.hu/nagyt/PPTMSC/7_Adatszerkezetek.pdf
- http://digitus.itk.ppke.hu/~monla/Iskolai/szig/prognyelv_2.doc
- http://www.cs.man.ac.uk/~pjj/cs212/fix.html
- http://hu.wikipedia.org/wiki/Objektumorientált_programozás
- http://aszt.inf.elte.hu/~fun_ver/2002/papers/tipuselm_oo.pdf
- http://people.inf.elte.hu/gt/oaf/pointer_mf.pdf
- Kozsik Tamás diái a „Programozási nyelvek: Ada” tárgyhoz: http://aszt.inf.elte.hu/~kto/teaching/ada/eloadas/
- Kozsik Tamás diái a „Java alkalmazások” tárgyhoz: http://aszt.inf.elte.hu/~kto/teaching/java/