4 Adatszerkezetek, adattípusok, adattárolás

Az informatikában az adatokat nem fizikai megjelenési formájukban tároljuk, hanem kódolva. **Kódoláson** azt értjük, amikor egy adatsorozatot, jelek sorozatával helyettesítjük, és a jelsorozatot tároljuk, továbbítjuk. A kódolás célja sokféle lehet. Az adatok érthetőbb vagy egyértelmű megjelenítése, tömörebb tárolása, gyorsabb átvitele, stb. A kódolandó adat félesége meghatározza azt, hogy hányféle jellel lehet kódolni az adatot. Gyakori eset, hogy egy már kódolt jelsorozatot továbbkódolunk, és így többszörös kódolás jön létre.

Ha az eredeti adatokat vissza akarjuk nyerni, akkor a kódolás fordítottjának, a **dekódolásnak** kell lezajlania. Ha egy adatsort többszörösen kódolunk, akkor a dekódolások sorozata a kódolás sorozatával fordított sorrendben zajlik le.

A programozási nyelvek az adatokat bizonyos keretek között tárolják.

A számítógépek fizikai felépítéséből következően az adatokat bitek sorozataként tárolják a memóriában. Tekintettel arra, hogy egy bit csak kétféle adatot tartalmazhat, ezért a biteket nagyobb egységekbe szervezik és a processzorok kényelmes módot adnak a nagyobb egységekben történő adatkezeléshez.

Bár az adatok tárolásának módja programnyelvenként, sőt a programnyelv implementációiként is változhat, ugyanakkor vannak olyan alapvető adatszerkezetek, amelyek a gépek tulajdonságai miatt azonosak.

A számítógépek az adatokat **byte**, **word** (2 byte = szó) vagy **duplaszó** (4 byte) hosszú egységekben kezelik. A processzorok típusától és a memóriák szervezésétől függően a háromféle adatkezelés sebessége más és más. Erre példa, hogy a nyolc bites processzorok a byte hosszú adatokat kezelték a leggyorsabban, a 80286-os processzorok a szó hosszúságú adatokat, míg a 386-os és annál fejlettebb Intel processzorok a duplaszó hosszú adatokat kezelik a leggyorsabban. Ha optimális sebességű programot akarunk írni, ekkor az adattípusok meghatározásánál erre is figyelni kell.

Amikor egy programozási nyelv adatainak tárolását vizsgáljuk, akkor feltételezzük, hogy van egy kellően nagy memóriánk, amely byte nagyságú egységekből áll. A byte-ok sorfolytonosan helyezkednek el a memóriában.

Az adatok tárolása során a számítógépet nem érdekli egy adat értelme, jelentése. A programozónak kell módot találnia arra, hogy az adatot jelentésének megfelelően kezelje. A számítógépes programok azonban nyilvántarthatják bizonyos formai módszerekkel az adatok egyéb paramétereit.

Az adatok jellemzői:

Minden adat rendelkezik **típussal**

Az adattípus meghatározza az adat lehetséges **értékét**. Az adattípus lehetséges értékeinek halmaza az **értéktartomány**.

Az adat által a háttértáron vagy a memóriában elfoglalt hely nagysága az adat **mérete**. Egyes adattípusok mérete eleve adott, más adattípusok mérete a definíció során alakul ki a rész adattípusok méreteiből következően, míg vannak olyan adattípusok, amelyeknek mérete a program futása közben dől el

Fontos az adat elhelyezkedése a memóriában. Ez az adat **címe**. A programok az adat címének ismeretében képesek hozzáférni az adat értékéhez, azt a megadott helyről kiolvassák és szükség esetén módosítják, illetve visszaírják a háttértárakra.

Az adat típusa egyértelműen meghatározza az adattal végezhető műveleteket (pl. numerikus adatokkal matematikai műveleteket lehet végezni, de szöveggel nem ...).

Egyes adattípusok mérete eleve adott, más adattípusok mérete a definíció során alakul ki a rész adattípusok méreteiből következően, míg vannak olyan adattípusok, amelyeknek mérete a program futása közben dől el.

Alap adattípusok – a rendszerekben előre definiált típusok. A rendszerekben általában van lehetőség az alap adattípusok és a korábban definiált típusok felhasználásával új típusok definiálására is.

Bővített adattípusok – A rendszerekben meglévő alap adattípusok felhasználásával létrejövő új típusok, amelyek az eredeti alaptípus valamilyen tulajdonságának a módosításával jönnek létre. Ilyen módosítás lehet pl. az értéktartomány szűkítése, bővítése.

Korábban már definiáltuk a **konstans** és a **változó** fogalmát. Amikor deklarálunk egy változót, vagy egy konstanst létrehozunk az ún. erősen típusos nyelvek esetén – ilyen a C, C++, Pascal, Visual Basic – a változónak egyúttal megadjuk a típusát is. Más nyelvek esetén a változó típusa futás közben, az első értékadáskor dől el. Vannak olyan nyelvek, ahol a változó az értéktől függetlenül változtathatja adattípusát – pl. PHP.

4.1 Az adattípusok osztályozása megszámlálhatóság szerint

A halmazelméletből ismert fogalom a megszámlálhatóság. Amikor egy halmaz elemeit hozzá tudjuk rendelni a pozitív egész számok halmazához, akkor mondjuk, hogy a halmaz **megszámlálható számosságú**. Ez a gyakorlatban azt jelenti, hogy egyesével végig tudok valamilyen módon lépkedni a halmaz elemein. Ilyen halmaz például az egész számok halmaza, a betűk halmaza, stb. Az ilyen adattípusokat szokás **felsorolási** adattípusoknak is hívni.

A valós számok halmaza azonban nem megszámlálható, mivel mindig tudok két olyan számot mondani, amelyek között végtelen számú valós szám található, és amelyek nem bejárhatók. Ennek a **halmaznak a számossága nem megszámlálható**. Ilyen adattípus a valós számok vagy lebegőpontos számok halmaza. Ezen a ponton meg kell állni, ugyanis beleütközünk az abba a ténybe, hogy a valós számítógépek fizikailag korlátosak minden szempontból, azaz végtelen számú értéket véges helyen nem tudunk tárolni! Kompromisszumot kell kötnünk, és azt mondjuk, hogy azok az adattípusok nem megszámlálható számosságúak, amelyek részére nem áll rendelkezésre megfelelő kapacitás. Persze ez egy kicsit sántít, de ez van.

4.2 Az adattípusok osztályozása bonyolultság szerint

Definíció

*Az **egyszerű adattípusokat** a rendszer egységként kezeli és nem bontható szét összetevőkre.*

Definíció

***Összetett adattípusok** más adattípusok valamiféle kombinációjaként jönnek létre. A rendszer nem feltétlenül tudja egyben kezelni az összetett adattípust, ebben az esetben a programozónak kell gondoskodni a feldolgozásról. Az összetett adattípusnak vannak önállóan is feldolgozható rész adatai.*

Az összetett adattípusokból is készíthetünk további összetételeket, így meglehetősen bonyolult adatstruktúrákat lehet kialakítani.

Az adattípusok helyett szokás az **„adatszerkezet”** kifejezést is használni.

4.3 Egyszerű adattípusok

Az egyszerű adatszerkezeteket minden számítógépes rendszer ismeri. Ilyenek a numerikus, karakter, logikai típusok. Az egyszerű adattípusokat a gépek processzor szinten, azaz gépi kódú utasításokkal tudják kezelni, ezért használatuk a lehető leggyorsabb programokat eredményezi.

4.3.1 Numerikus típusok

Tetszőleges alapú számok ábrázolása

A számokkal való műveletek során hozzászoktunk a 10-es alapú számrendszerhez, de a számítógépeknek mindegy, hogy mi a használt számrendszer alapja (elvileg). Technikailag célszerű olyat választani, amely illeszkedik a számítógépek belső lelkivilágához, ezért a legjobban elterjedt rendszerek a kettes (bináris), 16-os (hexadecimális) és a 8-as (oktális) számrendszer. A **számrendszer alapszáma** egyúttal **megadja a** szám leírásánál **használható jelek számát** is.

4.3.1.1 Kettes számrendszerbeli számok ábrázolása

A kettes számrendszer esetén a számokat 0 és 1 jelek sorozatával jelöljük. Annyi jelet írunk le, hogy a jelek száma kettő hatványaival legyen egyenlő (1, 2, 3, 8, 16, 32, 63 jel), Egy jelet egy bit-nek hívunk (**Bi**nary Digi**t**). Ennek alapján beszélhetünk 4, 8, 16 stb... bites számokról.

Egy szám átalakítása 10-esből kettesbe.
Elosztjuk a 10-es számrendszerbeli számot 2-vel, a **maradék** lesz a jobbról számított legutolsó bit bit, majd a hányadost osztjuk 2-vel és a legszélső lesz az utolsó előtti bit, stb...)

Az így kapott bitsorozatot kiegészítjük annyi nullával, hogy megfeleljen a kívánt tárolási méretnek. (8 bit, 16 bit, 32 bit, stb...)

Egy szám átalakítása kettes számrendszerből 10-esbe
A jobbról számított első bit jelenti a kettő 0. hatványát, a következő kettő 1. első hatványát, stb... Ha 1 van a megfelelő helyiértéken, akkor az eredményhez hozzáadom a megfelelő 2 hatványt.

00010101 = 0*128+0*64+0*32+1*16+0*8+1*4+0*2+1*1 =23

0 és 1 közötti 10-es számrendszerbeli számok bináris ábrázolása.
A számot elosztom 2-vel. A hányados lesz a kettedes pont utáni első jegy. A maradékot elosztom 2 következő hatványával, és így folytatom, amíg a megfelelő tárolási méretet el nem érem.

Osztandó /osztó	Hányados	Maradék
0,7 / 0,5	=> 1	0,2
0,2 / 0.25	=> 0	0,2
0,2 / 0,125	=> 1	0,075
0,075 / 0,0625	=> 1	0,0125
0,0125 / 0,03125	=> 0	0,0125
0,0125 / 0,015625	=> 0	0,0125
0,0125 / 0,0078125	=> 1	0,0046875
...		

Az így kapott szám: 0,10110001

Látható, hogy az eredmény nem pontos, mivel az utolsó osztás után is van maradék! Az utolsó bit értékét kerekíteni kell. Általában igaz, hogy az egyik számrendszerben megadott véges szám nem feltétlenül lesz végesek más számrendszerben.

4.3.1.2 Kettes komplemens számábrázolás

Az tetszőleges méretű előjeles egész számok ábrázolásának elve. A számot 2^n db bittel írjuk le, ahol n=8, 16, 32, stb. A legmagasabb helyiértékű bit az úgynevezett előjelbit. Az előjelbit 0 értéke jelenti azt, hogy a szám pozitív vagy nulla, az előjelbit 1-es értéke jelenti azt, hogy a szám negatív.

Hogyan állapíthatjuk meg egy 10-es számrendszerbeli számból a kettes komplemens alakját?

Vesszük a szám abszolút értékét. Ha a szám pozitív, akkor ez nem módosít semmit, ha negatív, akkor elhagyjuk az előjelet.

Ennek a számnak kiszámítjuk a bináris értékét. (korábban láttuk)

Az így kapott bitsorozatot kiegészítjük annyi nullával, hogy megfeleljen a kívánt tárolási méretnek. (8 bit, 16 bit, 32 bit, stb...)

Ha az eredeti szám pozitív volt, akkor végeztünk.

Ha az eredeti szám negatív volt, akkor a kapott bináris számban minden bit értékét felcserélem 0-ról 1-re és 1-ről 0-ra. Ekkor kapom az **egyes komplemens értéket.**

A végén hozzáadunk 1-et.

Pl.

Eredeti szám	22	=>	00010100
Egyes komplemens:		=>	11101011 + 1
Kettes komplemens:	-22	=>	11101100

4.3.1.3 Oktális számok ábrázolása (8-as számrendszer)

Az oktális számoknál a használható jelek 0,1,2,3,4,5,6,7 Az oktális számokat mindig hármasával csoportosítva írjuk le. A 10-es számrendszerből az alábbiak szerint számoljuk át a számot 8-asba. Az eredeti számot elosztjuk 8-cal, és a maradék lesz a legkisebb helyiértékű szám. A hányadost osztjuk megint 8-cal és a maradék lesz a következő jegy, stb...

196 / 8 => 24, maradék 4
24 / 8 => 3, maradék 0
3/ 8 => 0, maradék 3

Az eredmény: 304

A szokásos jelölése az oktális számoknak az, hogy balról kiegészítjük egy 0-val a számot, azaz 0304.

Visszafelé hasonlóképpen járhatunk el, mint a bináris ábrázolásnál.

Bináris számokat átalakítani oktálissá egyszerű, mivel 3 bit alkot egy oktális számjegyet, azaz:

0304 => 000 011 000 100, ami persze csak 8 bites lesz, azaz 11000100.

4.3.1.4 Hexadecimális számok ábrázolása (16-os számrendszer)

A hexadecimális számoknál a használható jelek 0,1,2,3,4,5,6,7 ,8,9,A,B,C,D,E,F. A hexadecimális számokat mindig négyesével csoportosítva írjuk le. A 10-es számrendszerből az alábbiak szerint számoljuk át a számot 16-osba. Az eredeti számot elosztjuk 16-tal, és a maradék lesz a legkisebb helyiértékű szám. A hányadost osztjuk megint 16-tal és a maradék lesz a következő jegy, stb.

196 / 16 => 12, maradék 4 => 4
12 / 16 => 0, maradék 12 => C
Az eredmény: C4

A szokásos jelölései a hexadecimális számoknak: 0xC4, #C4, $C4

Visszafelé hasonlóan járunk el, mint a bináris számoknál.

Bináris számokat hexadecimálissá és vissza konvertálni egyszerű, mivel 4 bit alkot egy hexadecimális számjegyet:

192 => 0xC4 => 0110 100

4.3.1.5 Tetszőleges alapú számrendszerek (nem törzsanyag érettségire)

Az x alapú számrendszer esetén a használt jelek 0-tól (x-1) –ig terjednek.

Ha a számrendszer alapja x, akkor 10-esből erre az alapra úgy konvertálunk, hogy elosztjuk az eredeti számértéket az alappal és a maradék lesz a számrendszerbeli szám utolsó jegye. A hányadost osztjuk megint és így tovább.
Visszakonvertálni úgy kell:

Ha van egy x alapú számom: $a_1,a_2...a_n$, jelsorozat az alábbi polinom értékét jelöli:

$a_1 *X^{n-1} + a_2 *X^{n-2}... + a_n *X^0$

4.3.1.6 Byte

A mérete 1 byte, értéke 0..255 közötti egész szám lehet (2^8 érték)

Ennek egy bővített adattípusa a rövid egész vagy shortint, vagy short, amelynek az értékei

–128..,0, ..+127 közötti egész értékek lehetnek.

A szám előjelét a legmagasabb helyiértékű bit előjele jelöli. Ha a bit értéke egy, akkor negatív számról vagy a nulláról van szó. 8 bites rendszerekben a leggyorsabb adatkezelést biztosítják

4.3.1.7 Word - Szó

Mérete 2 byte, értéke 0..65535 közötti egész szám lehet. (2^{16} db érték)

4.3.1.8 Integer - Egész

Mérete két byte, értéke –32768..,0,..+32767 közötti egész értéket veheti fel. (2^{16} érték)

A szám előjelét a legmagasabb helyiértékű bit előjele jelöli. Ha a bit értéke egy, akkor negatív számról vagy nulláról van szó. 16 bites rendszerekben a leggyorsabb adatkezelést biztosítják az Integer és Word típusok.

Lehetséges műveletek:

Összeadás	+	A + B
Kivonás	-	A – B
Egész típusú osztás	Div vagy /	A div B vagy A / B
Egész típusok osztásakor maradéka	Mod vagy %	A Mod B vagy A % B
Eggyel való növelés, csökkentés	++, --	A++, B--

Összehasonlításokban

< kisebb, mint, > nagyobb mint, <= kisebb egyenlő mint, >= nagyobb egyenlő mint,

== egyenlő, <> vagy # nem egyenlő relációk lehetnek.

A fenti típusok bővítései a longint (Pascal), long int, unsigned long. Ezek 4 byte hosszú, adatok, ugyanazokkal a feltételekkel, amelyek a fentiekben tárgyaltunk. (2^{32} érték)

4.3.1.9 Lebegőpontos számábrázolás

A 10-es számrendszerben definiált számok ábrázolása összetett feladat, ráadásul rendszerenként kissé eltérő lehet. Külön problémát jelent a 0 ábrázolása is és a negatív számok ábrázolása. Elöljáróban elmondom, hogy a lebegőpontos számokat gyakran 4 bájton, azaz 32 biten ábrázolják, amelyből 1 bit az előjelbit, 23 a mantissza és 8 bit a karakterisztika (kitevő). Az előjelbit értéke 1-ha a szám negatív, 0 ha a szám pozitív.

Egy tízes számrendszerbeli lebegőpontos szám ábrázolásához az alábbi folyamaton kell végigmenni:

átalakítjuk a számot kettes számrendszerbelivé, külön az egészrészt és külön a tört részt: pl. -486,17 => 111100110.001010111000010

Normalizáljuk a számot (A normalizálásnál addig toljuk el a bináris pontot, amíg a bináris ponttól balra nem kerül az első 1-es bit): 1.11100110001010111000010 x 2^8

Innen külön kezeljük a mantisszát és a karakterisztikát.

A mantissza legfelső bitje mindig egyes, hiszen a normalizálásnál így toltuk el a bináris pontot. Ezt a bitet elhagyjuk (!) és a mantissza elé írjuk az előjelbitet, ami a példánkban 1-es. 11110011 00010101 11000010 lesz a mantissza

A karakterisztikánál trükközünk egyet, a kitevő értékéhez hozzáadunk 128-at. Ennek az indoka az, hogy ezáltal – kitevő is pozitív számmá változna, tehát a kitevő 8 => 8+127 =135 => 10000111

A karakterisztikát az Intel processzoros gépeken mindig a mantissza előtt tárolják a memóriában, tehát a teljes szám így fog kinézni: 10000111 **1**1110011 00010101 11000010

Lehetséges műveletek

Összeadás	+	A + B
Kivonás	-	A – B
Osztás	/	A / B
Matematikai függvények.	sin, cos, tg, atg exp, log, …	
Kerekítés, törtrészképzés	Round, trunc	

Összehasonlításokban

< kisebb, mint, > nagyobb mint, <= kisebb egyenlő mint, >= nagyobb egyenlő mint,

== egyenlő, <> vagy # nem egyenlő relációk lehetnek.

A real bővítésének felelnek meg a double 8 byte, extended, long double 10 bytet ípusok.

Általában az összeadás, kivonás, szorzás, osztás esetén az értékek lehetnek kevert típusúak is, azaz egész típus és lebegőpontos is, de az eredmény mindig olyan típusú lesz, mint a legnagyobb helyet elfoglaló érték vagy lebegőpontos. Az olyan függvények, amelyek lebegőpontos értéket várnak, nem elégednek meg egész típusú paraméterekkel és fordítva.

4.3.2 Karakter típus, kódlapok, karakterkódolás

Definíció

*A **karakter** egy írásjelet tároló adattípus.*

A karakteres adattípus egy karaktere a 8 bites karakterkezelésnél 1 byte helyet foglal, azaz 256 különböző értéket vehet fel. Valójában a karakterek tárolásakor a számítógép egy byte információt tárol, de hogy az milyen karakternek felel meg – milyen betűnek – az a kódolási rendszerektől függ.

A PC-k világában az **ASCII** (American Code for Information Interchange) karakterkódolási rendszer terjedt el. Ez a kódrendszer 7 bites, azaz a kódrendszernek csak az első 127 karaktere definiált.

A 0 – 31 kódú karakterek nem látható karakterek. A képernyőn mozgó kurzort – alfanumerikus képernyőn – a nyomtató fejét és egyéb olyan jeleket definiáltak, amelyeknek nincsen látható formájuk, de azért fontosak. Néhány példát csak: 0 – nem látható, 1- ☺, 2 - ☻, 10 – soremelés, 13 – kocsi vissza (Enter), 27 – Esc, …

A 32 – Szóköz. 33-47-ig különböző írásjelek vannak, 48 –tól 58-ig a 0,1,2,…9 számjegyek, 58-tól 63-ig írásjelek, 64 - @ jel (kukac).

65-tól 90-ig az angol ABC nagybetűi, A, B, C, D,….Z-ig. 91-től 96-ig írásjelek, majd 97-től –122-ig angol ABC kisbetűi, a, b, c, d,….z-ig. 123-tól 127-ig megint írásjelek.

Sok olyan nyelv van, amely ezen kívül még más írásjeleket is használ. Ezeket eredetileg a felső 128 jel közé tették volna be, de ez nem volt elegendő. Ennek a problémának a megoldására találták ki, az un. kódlapokat.

A **kódlapok** az ASCII karakterkészlet egyfajta kiterjesztései. A különböző kódlapok esetén az alsó 128 karakter megegyezik, de a felső 128 karakter az illető kódlapban érvényes jeleket jelent. A korrekt megjelenítés érdekében a futtató hardvernek, az operációs rendszernek, a nyomtatónak és minden egyéb eszköznek képesnek kell a különböző kódlapok használatára, csak úgy lehet korrekt megjelenítést elérni.

A probléma véglegesnek tűnő megoldása a 16 bites **Unicode** kódkészlet használata. Ekkor egy karakter két byte helyet foglal el a memóriában és ennek megfelelően 65535 féle jelet lehet megjeleníteni. Ez már elegendő a kínai és a japán valamint az egyéb nem európai nyelvek jeleinek a megjelenítésére is. Hátránya, hogy a karakterek több helyet foglalnak el. A mai operációs rendszerek alapvetően alkalmasak ennek a kódrendszernek a használatára.

4.3.2.1 Szövegek tárolása – sztring adattípus

Ha a számítógépek memóriájában a karaktereket egymás után tároljuk le, és valamilyen módon megadjuk, hogy hol van a karakterlánc vége, akkor egy új adattípust, a **karakterláncot** vagy közkeletű nevén **sztringet (string)** kapunk. Kétféle karakterlánc kezelési módszer terjedt el.

Az adott hosszúságú sztringek kezelése – A Pascalban a karakterek maximálisan 254 byte hosszúak lehetnek, mivel a 0. Helyen álló karakter kódja adja meg a sztring hosszát. Ez a karakter egyébként sohasem jelenik meg a képernyőn, de a program ennek alapján tudja megállapítani a sztring végét. Ennek az a hátránya, hogy nem lehet tetszőlegesen hosszú a karakter. A Pascalban előre meg kell mondani, hogy a karakter milyen hosszú lesz.

0 végű sztringek kezelése – A C nyelvben és annak minden folyományában a 0 végű karakterkezelés terjedt el, ami azt jelenti, hogy a sztring addig tart, amíg nem nulla karakterek jönnek egymás után. Ennek előnye, hogy elvileg tetszőleges hosszúságú, a gyakorlatban maximum 65535 karakter hosszú stringeket lehet kezelni. A hátránya, hogy azokban az esetekben, amikor szükséges tudni a sztring hosszát egy rutinnak végig kell szaladni a sztring elejétől a 0 kódú karakterig és meg kell számolnia a karaktereket – ez kis lassulást okoz a

sztring műveleteknél. Megjegyzendő, hogy a Borland Pascal 7-es verziójától létezik a **Pstring** nevű adattípus, amely 0 végű karakterkezelést tesz lehetővé a Borland Pascal nyelvekben.

A karakterekkel nem lehet semmiféle matematikai műveletet végezni. Vannak azonban alapvető műveletek, amelyeket minden programozási nyelvben el lehet végezni. A műveletek általában függvény alakban léteznek és a különböző nyelveken más és más a megvalósításuk is. Azt is el kell mondani, hogy általában azok a műveletek, amelyek karakterre alkalmazhatók, alkalmazhatók sztringekre is.

A fenti adattípusokkal lehetséges műveletek:

Összefűzés	+	'A' + 'B' => 'AB'
Karakter kódja	ASCII(k), ORD(k)	ASC('A') => 65
Kód alapján katakter	CHR(szám)	CHR(66) => 'B'
Sztring bal oldala	Left(sztring, szám)	LEFT('ABC',1) => 'A'
Sztring jobb oldala	Right(sztring, szám)	Right('ABCD',2) => 'CD'
Sztring középső karakterei	Mid(sztring, szám, dbszám)	MID('ABCDE',2,3) => 'BCD'

Összehasonlításokban

A karaktereket ugyanazokkal a jelekkel hasonlítjuk össze, mint a karakterláncokat,

< kisebb, mint, > nagyobb mint, <= kisebb egyenlő mint, >= nagyobb egyenlő mint,

== egyenlő, <> vagy # nem egyenlő relációk lehetnek.

Mindig balról jobbra kezdi a rendszer karakterenként az összehasonlítást és az első különbségnél már el is dönti az eredményt. Ha két karakterlánc ugyanazt tartalmazza végig, de az egyik rövidebb, mint a másik, akkor az a „kisebb".

Olyan adatok, esetén, amikor az adatokat több különböző típusú operációs rendszerből kezelik célszerű az adatokat karakterekként tárolni és lehetőleg az ASCII kódrendszert felhasználni rá.

Megjegyzés:

A Pascal és a C nyelv sztringkezelése között van különbség. A Pascalban a programozási nyelv része a sztrinkezelő függvények és eljárások sorozata, a C-ben viszont külön könyvtári függvények vannak, amelyeket a programhoz kell szerkeszteni.

4.3.3 Logikai típus

A logikai adattípus két értéket vehet fel, Igaz, vagy Hamis értékeket. Ez a két érték különböző programnyelveken lehet a True-False, Yes-No, Igen-Nem, Nem nulla – Nulla értékpárosok valamelyike.

Bár a logikai adatokat elvileg egy bit is egyértelműen jelzi, de a gépek bináris felépítése és a gépi kódú utasításkészletek mindig egy byte méretű adatként kezelik.

A logikai adatokra lehet alkalmazni a következő műveleteket:

<, >, <=, >=, <> Logikai És, Logikai Vagy, Logikai Kizáró Vagy, Tagadás.

A szokásos jelölések – gyakorlatilag a legtöbb programozási környezetben:

<, >, <=, >=,<>, #, AND, OR, XOR, NOT.

4.3.4 Mutatók, pointerek

Definíció

A mutató olyan változó, amely a memóriában lévő adat memóriacímét tartalmazza.

A memóriában tárolt adatoknak mindig van címük. Azokat a változókat, amelyek más adatok memóriacímét tartalmazzák, mutatóknak vagy pointereknek nevezzük. A pointerek az egyes programozási nyelvekben és operációs rendszerekben nem kezelhetők egységesen, nem egyforma a méretük sem, de egy adott operációs rendszer, memóriamodell és fejlesztőeszköz esetén pontosan meghatározhatók a pointerek méretei.

Ha az adatok egy 64Kb-os memóriaterületen elférnek, akkor elegendő a pointernek csak a terület elejéhez viszonyított eltolást (offszet) tárolni. Ebben az esetben a pointer méretére elegendő csak 2 byte. Ha nagyobb memóriaterületet akarunk megcímezni, annak megfelelően kell a pointerek méretét növelni, általában 4 byte vagy 8 byte lesz a pointer mérete.

Definíció

A mutató (pointer) típusa annak az adatnak vagy változónak a típusából ered, amilyen adatra vagy változóra a pointer mutat.

Ilyenkor a pointernek más típusú adatra vagy változóra nem mutathat.

A Pascal és a C, mint két alapvető programozási nyelv kissé más szintaktikával jelöli a mutatókat és a végezhető műveletek köre is más és más.

A PC-k esetén a pointerek általában segmens:offset, módon, 4 bájton tárolódnak. Az Intel processzorok tulajdonságai miatt a tárolás alsó két bájtja az offszete, a felső két bájt a szegmens tartalmazza.

Pascal nyelv	
v változó címének letárolása p pointerbe	P:=@v vagy p:=Addr(v)
Hivatkozás a mutatott p változóra	p^
Értékadás **v** változónak pointer segítségével	p^ := 133
A „sehová sem mutató” pointer konstans	Nil vagy Ptr(0,0)
Összehasonlítások	<>,=
p pointer offszetje	Ofs(p)
p pointer szegmense	Seg(p)

A C nyelv lehetőségei a pointerekkel való műveleteket nagyon elősegítik, de a pointerek használata a programokat kissé áttekinthetetlenné teheti. Ha az ilyen programot jól írjuk meg, akkor a pointerek használata hatékonyságjavulást okoz.

C nyelv	
v változó címének letárolása p pointerbe	p = &v
Hivatkozás a mutatott **v** változóra	*p
Értékadás **v** változónak pointer segítségével	*p = 133
A „sehová sem mutató” pointer konstans	Null
Összehasonlítások	==, !=
pointer által mutatott változó módosítása	*p = *p + 33

4.3.5 Megszámlálható és nem megszámlálható adattípusok

Az eddig tárgyalt összes egész típusú adat, a karakter típus és a logikai típus egyben megszámlálható típus is. A megszámlálható típusokat lehet létrehozni úgy is, hogy felsoroljuk a típust alkotó lehetséges értékeket. A megszámlálható adattípusok egyes elemei bitmintaként tárolhatók, ezért általában a megszámlálható adattípusok tárolási egységei a byte 2 hatványai.

A fenti adattípusokkal lehetséges műveletek:

Első adat	Típus(0)	0.
Adattípus következő adata	Succ(adat)	Succ(’A’) => ’B’
Adattípus előző adata	Pred(adat)	Predd(199) => 198

4.3.6 Konverziók

A különböző nyelveken az egyes adattípusokat más és más függvényekkel lehet átkonvertálni. Erre azért van szükség, mert az egyes programozási nyelvek a különböző típusokat másképpen kezelik. A konverzió alól a C nyelv bizonyos mértékig mentesíthető, mivel a C-ben a rövid egész típust lehet karakternek is tekinteni, továbbá sztringet lehet rövid egészek tömbjének tekinteni stb… Az adatkonverziók leggyakoribb eljárásai:

Karakter kódja	ASCII(k), ORD(k)	ASC(’A’) => 65, ORD(’B’) => 66
Kód alapján katakter	CHR(szám)	CHR(66) => ’B’

Realból integer	Round(Real)	Round(1.321) => 1
Integerből Real	Int(egész típusú szám)	Int(344) => 344.00
Sztringből szám	VAL(sztring, szám, dbszám)	VAL('123.34', változó, hibakód) 'A Pascalban ez úgy működik, hogy a hibakód megmutatja, hogy a konverzió hibátlan volt-e'
Számból Sztring	STR(szám)	STR(1876.01) => '1876.00'

4.3.7 Globális- és lokális adatok kezelése, az adatok láthatósága

Amikor egy programban adatokkal dolgozom általában nincsen szükségem a programban felhasznált bármelyik adat elérésére. A jól strukturált programokban egy eljárás vagy függvény jól specifikálhatóan csak bizonyos adatokkal dolgozik. A feldolgozandó adatokat vagy az őt hívó eljárástól kapja meg, vagy a függvényben – eljárásban keletkezik és esetleg az őt hívó eljárásnak adja vissza, vagy amikor kilép a programrészletből, akkor elenyészik, mert nincsen rá szükség.

A BASIC programozási nyelv eredetileg globális adatokkal dolgozott csak, azaz minden adatot mindig el lehetett érni a program minden pontjáról. Bonyolultabb programok esetén az adatok következetes kezelése nehézkes, sőt csaknem megoldhatatlan.

A probléma megoldására vezették be a **globális** és a **lokális adat/változók** fogalmát.

Egy **adat lokális** egy programegységre nézve, ha abban a programegységben az adat elérhető, esetleg módosítható, míg a programegységet hívó programegységekből az adat nem látszik. Általában az ilyen adat akkor jön létre, amikor elkezdi végrehajtani a programegységet a program és akkor szűnik meg, ha kilép belőle.

Egy **adat globális** egy programegységre nézve, ha az adat már létezik akkor, amikor elkezdődik az egység végrehajtása. Nyilván ha egy adat globális egy programegységre nézve, akkor az egységből meghívott programrészekre is globális az adat.

Az **adatok láthatóságának** kérdése összefügg az adatok globalitásával is.

Általában ha egy **adat globális** egy programegységre nézve, akkor látható is, de ez nem minden programozási nyelven igaz.

A Pascalban csak azok a globális adatok láthatók, amelyek a programszövegben előrébb vannak deklarálva, vagy speciális kulcsszóval utasítjuk a fordítót, hogy tegye láthatóvá máshonnan is. A C nyelven a header fájlokban kell megadni azoknak a globális változóknak a definícióját, amelyeket más eljárásokban látni szeretnénk.

Ha egy eljárásban ugyanolyan nevű változót definiálunk, mint amilyen egy globális változó neve, akkor a **lokálisan definiált** változó válik láthatóvá, eltakarja a globális változót. Ez még akkor így van, ha a két változó típusa nem egyezik meg. Bizonyos nyelvek ebben az esetben fordításkor hibaüzenetet adnak, de ezzel most nem foglalkozunk. Ha kilépünk a kérdéses eljárásból, akkor természetesen megint az eredeti globális változó válik Ha a lokális adat létrejöttekor meghívunk egy eljárást és egyébként a nyelv szintaktikája engedi, akkor a meghívott eljárásban is látható az adat, hiszen arra az eljárásra nézve a kérdéses adat globális.

Vannak olyan speciális nyelvek, amelyek képesek létrehozni egy beágyazott – azaz valahonnan meghívott - eljárásban is a teljes programra nézve globális adatokat. Ilyen a Clipper programozási nyelv.