A programozási nyelvek adat típusairól

A számítógépben zajló folyamatok megfelelő lefuttatásához különböző információkra van szüksége a folyamatokat vezérlő processzornak. Az információk a számítógép operatív memóriájában adatok formájában állnak rendelkezésre. Egy 1 GB méretű RAM pontosan 1024 * 1024 * 1024 = 1 073 741 824 byte, illetve 8 * 1 073 741 824 = 8 589 934 592 bit, azaz körülbelül 8,6 milliárd elemi cellát tartalmaz. Az alábbi ábrán ennek a hatalmas méretű operatív tárnak egy parányi részlete látható egy lehetséges állapotban:

0	1	1	1	1	1	0	1	0	0	1	1	1	0	1	1	0	1	1	0	1	0	1	0	1	1	1	1	1	0
0	1	0	0	1	1	0	1	0	0	1	1	1	0	1	0	1	0	1	1	1	0	0	0	1	1	0	1	1	0
1	1	0	1	1	0	0	1	1	1	1	0	1	1	0	1	0	0	1	1	0	1	1	0	1	0	1	1	0	1
1	0	1	0	1	0	1	1	0	1	0	1	0	1	1	1	0	1	1	0	0	0	1	1	0	1	1	0	1	1
1	1	0	1	1	0	0	1	1	1	0	0	1	1	1	1	0	1	0	1	1	1	1	0	1	1	0	0	1	0
1	1	0	1	1	0	0	1	1	1	1	1	0	0	1	1	1	0	1	1	0	1	1	0	1	1	0	1	0	1
0	1	1	0	1	1	0	0	1	1	1	1	0	0	1	1	1	1	0	0	1	1	0	1	1	0	1	1	0	0
0	0	1	1	0	1	1	0	0	1	1	0	1	1	0	1	1	0	1	1	1	1	0	1	0	0	1	0	1	1
0	0	1	0	1	0	1	0	1	1	1	1	0	1	1	0	0	1				0	1	1	0	0	1	1	1	0
1	1	0	1	1	0	0	1	1	1	1	1	1	0	0	1	1	0	1				1	1	0	0	1	1	1	0
1	1	0	1	1	0	0	1	1	1	1	0	0	1	1	0	1	0	0	1	1	1	0	1	1	0	0	1	1	1
1	1	0	0	1	1	1	1	1	1	0	0	1	1	0	0	1	1	1	0	0	1	1	0	1	1	0	0	1	1
1	1	0	0	1	1	1	1	0	1	1	0	0	1	0	0	0	1	1	0	1	1	0	1	1	1	1	0	1	0
0	0	1	1	0	0	1	1	0	0	1	1	0	1	1	0	0	0	1	1	0	1	1	0	1	1	1	1	0	1
1	1	0	0	1	1	1	1	0	1	1	0	1	0	1	1	0	0	1	1	0	1	1	1	1	0	1	0	0	1
0	0	1	1	0	0	1	1	0	0	1	1	0	1	0	1	0	1	1	1	1	1	0	0	1	1	1	0	0	1
				1	0	0	1	1	1	1	1	1	1	0	0	1	1	1	1	1	1	0	0	1	1	1	1	1	0
0	1	1	1	1	1	0	1	1	1	1	0	1	1	0	0	1	1	1	1	0	1	1	0	0	1	1	1	1	0
0	1	0	0	1	1	0	1	1	1	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0	1
					1	0	0	1	1	1	0	0	1	0	1	0	0	1	1	1	1	0	1	0	1	1	1	1	0
1	1	0	0	1	1	1	1	1	1	0	0	1	0	1	1	0	0	1	1	1	1	1	1	0	0	1	1	1	1
1	1	0	0	1	1	1	1	0	1	1	0	1	0	1	1	0	0	1	1	1	1	0	1	1	0				
0	0	1	1	0	0	1	1	0						0	0	1	1	0	0	1	1	0	0	1	1				1
1	1	0	0	1	1	1	1	1						1	1	1	0	1	1	0	1	1							
1	1	0	0	1	1	1	1	0			1	1	0	1	1	1	1	0	0	1	1	0							
0	0	1	1	0	0	1	1	0						0	1	1	0	1	1	1	1	0							
1	1	0	0	1	1	1	1	0						1	0	0	1	1	1	1	0	1	0	1	1	1	1	1	0
0	1	1	0	0	1	1	0	1	1	1	1	1	0	0	1	1	0	0	1	1	0	1	0	1	1	0	1	1	0
1	0	1	0	1	1	1	1	0	1	1	0	0	1	1	0	1	0	0	1	1	1	0							
1	0	0	1	1	1	1	1	1	1	1	0	0	1	1	1	1	0	1	1	0	0								
1	0	0	1	1	1	0	0	1	1	1	0	0	1	1	1	1	0	1	1	0	0	1	1	1	0	0	1	0	1
1	0	0	1	1	1	1	1	0	0	0	1	1	0	0	1	1	0	0	1	1	0	1	1	1	1	1	0		
1	1	0	0	1	1	1	1	0	1	1	0	0	1	1	1	1	0	1	1	1	0	0	1	1	1	1	0	1	0
0	1	1	0	0	1	1	0	1	0	1	1	0	0	1	1	0	1	1	0	1	1	0	0	1	1	0	1	1	0
1	0	1	0	1	1	1	1	0	1	0	1	0	1	1	1	1	0		1	0	1	0	1	1	1	1	0		
1	0	0	1	1	1	1	1	1	1	0	0	1	1	1	1	1	1		1	0	0	1	1	1	1	1	1		

Találós kérdés:
Egy *szam* nevű változó számértékét szeretnénk kiolvasni a fent ábrázolt cellatartományból. Hol található a szám kódja? *

Válasz:
Egy érték gyors és hibamentes kiolvasásához két információra van szükségünk:
1.) melyik cellában kezdődik a kívánt adatot leíró *0 -1* sorozat;
2.) hány egymást követő cellát foglal el az adat kódja.

*Válasz:
Egy érték gyors és hibamentes kiolvasásához két információra van szükségünk:
1.) melyik cellában kezdődik a kívánt adatot leíró *0 -1* sorozat;
2.) hány egymást követő cellát foglal el az adat kódja.

1.) A RAM minden egyes cellájához tartozik egy a cellát egyértelműen azonosító cím. Nincs más dolgunk, mint megjegyezni az adat eltárolásakor a memóriamenedzser által kiosztott címet. Egy vacsorával egybekötött esküvői mulatságon is meg kell jegyeznünk, hogy pl. a 26-os asztalnál ülünk (*bázis cím*), továbbá mondjuk az 5-dik széken (*offset cím*), egyébként előfordulhat, hogy a táncról visszatérve más poharába iszunk bele! :(

Ha sok adatot – esetleg több ezret – kell gyors egymásutánban kinyernünk a memóriából, nyilván nem megoldás az, hogy egy papírlapra írogatjuk a címeket. A gyorsaság érdekében a címek tárolását is elektronikus formában kell megvalósítani, méghozzá magában a memóriában. Definiálunk egy jól meghatározott helyet a memória belsejében és "táblázatos" formában ideírjuk az egyes adatok címeit. A könnyebb visszakereshetőség érdekében minden címet egy rövid betűsorozattal (névvel) azonosítunk. Ezt a formális betűsorozatot, mint változónevet fogjuk használni a hivatkozásokban.
Egy építészeti tervezőprogramban például éppen szükségünk van a tervezett épület ***h*** magasságára. Ekkor a processzor a címtáblázat elejére ugrik és elkezdi keresni az egymást követő bejegyzések között a "***h"*** bejegyzést. Amikor megtalálta, kiolvassa a bejegyzés mellett található, a változó aktuális helyére vonatkozó címet, végül a címre ugrik és kiolvassa onnan ***h*** értékét, mindezt néhány tízezred másodperc alatt.

2.) Újabb probléma merül fel azonban a kezdőcímre ugrás után: hány bitet olvassunk ki?
A problémát a korábban ábrázolt operatív tár egy még kisebb részletén elemezzük. Tegyük fel, hogy kezdőcímünk a szürkével jelzett cellára mutat.

1	1	0	0	1	1	1	1	0	1	1	0	1	0	1	1
0	0	1	1	0	0	1	1	0						0	0
1	1	0	0	1	1	1	1	1						1	1
1	1	0	0	1	1	1	1	0			1	1	0	1	1
0	0	1	1	0	0	1	1	0						0	1
1	1	0	0	1	1	1	1	0						1	0
0	1	1	0	0	1	1	0	1	1	1	1	1	0	0	1
1	0	1	0	1	1	1	1	0	1	1	0	0	1	1	0
1	0	0	1	1	1	1	1	1	1	1	0	0	1	1	1
1	0	0	1	1	1	0	0	1	1	1	0	0	1	1	1
1	0	0	1	1	1	1	1	0	0	0	1	1	0	0	1
1	1	0	0	1	1	1	1	0	1	1	0	0	1	1	1
0	1	1	0	0	1	1	0	1	0	1	1	0	0	1	1
1	0	1	0	1	1	1	1	0	1	0	1	0	1	1	1
1	0	0	1	1	1	1	1	1	1	0	0	1	1	1	1

Olvassunk ki először nyolc bitet: *01100110*.
A kapott érték tízes számrendszerben: **102** (toronyház).
Most olvassunk ki két byte-ot: *01100110 11111001*.
A kapott érték tízes számrendszerben: **26361** (égig érő űrobjektum?).

A kódoláshoz használt byte-ok számát a változóhoz rendelt ***típus*** információval tudjuk meghatározni. Ha adatunk ***Byte*** típusú, ez arra utal, hogy egyetlen byte-on kódoltuk, így értéke $00000000_2 = 2^0-1 = \mathbf{0}_{10}$ és $11111111_2 = 2^8-1 = \mathbf{255}_{10}$ között változhat. Ha adatunk ***Integer*** típusú hasonló meggondolások alapján értéke ***0*** és ***65535*** között változhat. A programok forráskódjában a fentiek apróbb eltérésektől eltekintve lényegében a következő módon jelennek meg:

```
h      : Byte;
szam : Integer;
```

Minden változót definiálni kell a használata előtt. A fordító (*compiler*) elhelyezi a változók táblázatában a nevet, a hozzátartozó típus információval együtt (definíciós mozzanat), illetve lefoglalja a memóriában a típus információ alapján a megfelelő hosszúságú memória területet, és társítja a névhez a kezdőcímet (deklarációs mozzanat). A lefoglalt cellák ekkor még üresek vagy rosszabb esetben maradék információkat tartalmaznak a korábbi használatuk következményeként. A memória terület releváns feltöltése az értékadásnál történik meg (inicializációs mozzanat):

```
h := 102;
```

Összetett típusok

Ha több egymással szorosan összefüggő adatot szeretnénk tárolni az operatív tárban, célszerű ezeket nem külön-külön, a memóriában véletlenszerűen szétszórva tárolni, hanem egyetlen adategységként. Ezzel jelentősen csökkenthetjük a hozzáférési időt. Példaként egy ***record***okat tartalmazó ***tömb***öt definiálunk. Tegyük fel, hogy egy biztosítótársaság adatbázisban szeretné tárolni ügyfeleinek nevét, az ügyféllel kötött biztosítás kötvényszámát, illetve a kötvénydátumot, azaz a szerződéskötés dátumát. Több tízezres ügyfélkör esetén célszerű megtervezni és összefogni az adatstruktúrát. Az ügyfél nevét egy ***string***ben fogjuk tárolni, és rendelkezünk arról is, hogy a string 30 karaktert tárolhasson. A kötvényszámot ***Longint*** típusú számként, 4 byte-on fogjuk tárolni, a dátumot pedig 8 karakterből álló stringként. Ezt a következő definíciókkal valósíthatjuk meg egy program kódjában:

```
nev:        String[30];
kotvszam:   Longint;
kotvdatum:  String[8];
```

Az adatokat az említett célszerűségi szempontok miatt összefogjuk egy record struktúrában:

```
kotes    =    record
                begin
                 nev:          String[30];
                 kotvszam:     Longint;
                 kotvdatum:    String[8];
                end;
```

Ezzel, valójában egy új, saját típust definiáltunk. Ettől kezdve egy új változónak már ezt a típust is felajánlhatjuk az előre definiált, beépített típusok mellett:

biztositas: kotes;

Ezt követően az adatok eléréséhez csak hivatkoznunk kell a record egyes mezőire: *biztositas.nev*, *biztositas.kotvszam*, *biztositas.kotvdatum*. Ekkor a *nev* hivatkozás helyett használandó *biztositas.nev* hivatkozás apró formális eltérésén túl lényeges különbség a korábbi deklarációhoz képest, hogy most biztosan a memória egymást követő celláiban lesznek az összetartozó adatok és nem szétszórtan. Ez az adatstruktúra 30+4+8=42 egymást követő byte-ot fog elfoglalni az operatív tárban. Foglaljunk le ezután a memóriában egy 10 000 elemű tömböt, amely 10 000 ügyfél adatait lesz képes tárolni:

ugyfel : array[1..10000] of kotes;

Ezután például a 604-ik ügyfél kötvényszámát a következő hivatkozással lehet elérni:

ugyfel[604].kotvszam.

Ekkor a processzor az *ugyfel* névvel azonosított tömbváltozó kezdőcímére ugrik, mint báziscímre, majd onnan továbbugrik (604-1)*42 + 30 + 1 byte-ot. Ekkor éppen a 604-ik ügyfél kötvényszámát tároló négy egymást követő byte kezdő byte-ján áll a memóriában. Négy byte-ot kiolvasva így hozzájuthatunk a kívánt adathoz. A számításban szereplő két kulcsszámhoz (42 byte - egy record mérete; 30 byte - a record első mezőjének hossza) a típusinformációs bejegyzés segítségével, azaz a változót leíró ”típustérkép” alapján juthatott a processzor.

Objektumok

Tételezzük fel, hogy az előbbi példában említett biztosító társaság a különböző negyedévekben kötött biztosítások esetén különböző szorzókat alkalmaz, ezért gyakran szükségessé válik, hogy hozzáférjünk az egyes szerződések hónap adatához. Írhatunk egy saját függvényt, amely képes kinyerni a 8 karakterből álló dátumból az ötödik és hatodik karaktert, azaz a hónap értékét és ennek alapján meghatározza az aktuális negyedévet:

```
függvény Negyedév( n ): Byte;
Negyedév=EgészOsztás(SzövegbőlSzám(Ball(Jobb(ugyfel[n].datum,4),2))-1,3)+1;
függvény vége

//n : ügyfélkód  //Byte : a függvény visszatérési értékének típusa
```

Nézzük a számítás menetét egy példán keresztül:

ugyfel[n].datum =

2	0	1	4	1	2	0	6

Jobb(*ugyfel[n].datum*, 4)=”1206”

Ball(”1206”, 2)=”12”

SzövegbőlSzám(”12”)=12

EgészOsztás(12-1, 3) + 1=4 //negyedik negyedév

Tovább vihetjük korábbi gondolatmenetünket az összetartozó adatokkal kapcsolatban, hiszen a *Negyedév* függvényünk kódja végső formájában szintén 0-ák és 1-esek sorozata valahol az operatív tárban. Mivel ezt a függvényt kifejezetten egy egyedi adatstruktúrára definiáltuk, azaz a függvény adatstruktúra-specifikus, így csak az említett adatstruktúrán van értelme használni. Célszerű ezért a függvényt rögtön az adatstruktúra mellé helyezni és egységbe zárni az adatokkal.(encapsulation) Ily módon eljutottunk egy újabb típushoz, az ***objektum***hoz. Természetesen egy objektumnak is van ”típustérképe”, és ez az objektumot leíró kód kiemelt elemeihez rendelt memória koordinátákból áll. A fordító magát az objektumot is és az objektum jellemző strukturális elemeit is, például az adatokat és függvényeket egyedi kezdőcímmel látja el. A belső strukturális elemeknek a tartalmazó objektum kezdőcíméhez képest megadott offset címei ugyanakkor általában nem nyilvánosak, így azokat közvetlenül nem tudjuk meghívni. A fordító az objektum belsejében egy nem dokumentált (rejtett) kezdőcímmel ellátott helyen nyilvános nevekből álló index táblázatot készít és a nyilvános nevekhez hozzárendeli a belső adatok és eljárások kezdőcímeit. Kicsiben tehát ugyanazt valósítja meg, mint nagyban a memóriamenedzser. Minden az objektummal kapcsolatos futtatási kérelmet az objektum ezután saját indextáblájához irányít. Ha a táblában van olyan formális név, amely megegyezik a futtatási kérelemben szereplő névvel és annak típusával, metódus esetén paraméterlistájával és a paraméterlista egyes elemeinek típusával, a kérelmet végrehajtja. Egy objektumot tehát csak akkor lehet értelmes módon használatba venni, ha készítője dokumentálja az objektumban nyilvánosan elérhető változókat és metódusokat. A bevezetésben mutatott operatív tár részlet is lehet egy objektum kódja, de ebben a formában, dokumentáció nélkül semmire sem használhatjuk.

Példa:

A fentiekre egy szép példát a Java nyelvből hozunk. A Java nyelvben az egyébként egyszerű típusnak minősülő String típusból, amely karakterek közönséges sorozatát, azaz szöveget képes tárolni, objektumot készítettek a fejlesztők. A String objektumban helyet kaptak mindazok a metódusok is, amelyek a jellemzően előforduló szöveg manipulációs műveletek végrehajtásáért felelősek. Ilyen metódus például a *substring* metódus, amely részstringet ad vissza, vagy a *toUpperCase* metódus, amely nagybetűssé alakítja az objektum szövegtartalmát. Adjunk egy *szo* nevű string objektumnak értéket, és legyen ez ”számítás”:

szo =”számítás”;

Az alábbi kódrészlet az ”ÁMÍTÁS” értéket generálja:

szo.substring(2,8).toUpperCase( );

// itt az Excelben megszokott f(g(x)) formalizmus helyett **.**g(x)**.**f(x) –et használunk; Excel szerűen: *uppercase(substring(szo;2;8))*

Mivel az értékadás művelete, azaz egy változó inicializálása alapértelmezett művelet, ezért, ahogy az fentebb is látható, a fejlesztők arra törekedtek, minél egyszerűbben, a megszokott hagyományos formalizmussal lehessen a string objektumnak értéket adni (*szo*=”számítás”;). Ebből azonban téves lenne levonni azt a következtetést, hogy a *szo* névvel azonosított objektum-változó a kezdőcímétől kezdve egyesével tölti be az értékként kapott karaktereket a memóriába, ahogy azt a korábbi típusoknál láttuk. Ha ugyanis egy feltételes elágazásban meg akarjuk vizsgálni, hogy még mindig ”számítás”–e a tartalma a változónknak, (és nem ”ÁMÍTÁS”) és a hagyományos módon kódolunk, hibaüzenetet kapunk!

```
If (szo = = ”számítás”) then  // HIBA!          // = = : egyenlő –e ?
      {…}
```

A *szo* változó már egy összetett, metódusokat is tartalmazó objektumot képvisel azaz nem lehet egyenlő egy egyszerű karaktersorozattal, még akkor sem ha ”értékük” azonos! Helyette a következő kódot kell használnunk:

```
If (szo.content = = ”számítás”) then
      {…}
```

A kód árulkodik arról, hogy meg kellett hívni egy rejtett, belső offset címhez tartozó, de a fejlesztő által értelemszerűen nyilvánossá tett *content* nevű metódust, hogy hozzáférhessünk az objektum értékéhez. A *content* metódusnak kellett gondoskodnia a nem nyilvános objektum-típustérkép alapján az objektum érték-tartalmának felkutatásáról és meghívásáról.