sebességeknél a rakéta hatásfoka kicsiny, ezért nem előnyös a rakéta autókban és más, viszonylag lassú mozgásoknál. Nagy sebességeknél is csökken a rakéták energetikai hatásfoka, azonban ez a körülmény nem szól a rakéták alkalmazása ellen, amíg nincs más a testek nagy sebességekre való felgyorsítására.

A kozmikus sebsségek elérése érdekében az űrhajókat függőleges irányban indítják fel (a légkörben való minél előbbi túljutás miatt), majd a sebesség irányát közel 90°-kal megváltoztatják.

Nagy sebsségek üzemanyag spórlással úgy érhetők el, hogy többlépcsős rakétákat használnak; a lépcsők tartályai az üzemanyag elégetése után fokozatosan leválnak, így az üres tartályok gyorsítására üzemanyag már nem fordítódik.

Végül megemlítjük két „titánnak" a nevét, akik az űrhajózás gyakorlati megvalósításaiban játszottak kimagasló szerepet: Wernher von Braun-t, aki többek között megtervezte a gigászi Szaturnusz V. típusú rakétát és Szergej Pavlovics Koroliovot, az első Szputnyik a Föld körüli pályára állítására használt rakéta főkonstruktörét.

Ferenczi János
Nagybánya

Szerk. megj.: A modern rakétatechnika megalapozója, az ameriaki Jet Propulsion rakéta hajtóművek kutatólaboratóriumának létrehozója és vezetője, a budapesti születésű Kármán Tódor (1881-1963), az ő nevét is érdemes megemlítenünk.

Borland Delphi - az Object Pascal nyelv

A Delphi fejlesztői környezet színfalai mögött az Object Pascal nyelv áll. A nyelv számos olyan újdonságot tartalmaz, amelyek biztosítják, hogy a Pascal nyelv alkalmas legyen Windows–alkalmazások fejlesztésére illetve olyan megoldásokkal szolgálnak, amelyek hatékonyabb programkódhoz vezetnek. A következőkben megpróbáljuk összefoglalni mindazon változásokat, újdonságokat, amelyeket az Object Pascal nyelv hozott.

Új típusok

A Windows fejlesztői környezet filozófiája nagyméretben befolyásolta a típusok kialakítását. Az egész típusok új elemekkel bővültek, lehetővé téve, hogy a fejlesztői környezet 16- vagy 32 bites mivoltát kihasználják. Megjelenik a *Cardinal* típus, amelynek mérete függ a rendszertől (26- vagy 32 bit). A logikai típus pedig a rendszer igényeinek megfelelően többféle méretben áll rendelkezésünkre: *Boolean* (*false..true*, 8 bites), *ByteBool* (*false..true*, 8 bites), *WordBool* (*false..true*, 16 bites), *LongBool* (*false..true*, 32 bites).

Változások a paraméterátadásban

Az Opject Pascal nyelv paraméterátadása is kibővült. Számos olyan új lehetőséget tartalmaz, amely hatékonyabb kód fordításához vezetnek.

A Pascal nyelvben megszokott kimeneti (*var*), illetve bemeneti paraméterek mellett lehetőség adódott ***konstans*** (*const*) paraméter átadásra is. A konstans paraméter értéke az eljáráson vagy a függvényen belül nem változtatható meg. A konstans paraméterekről a fordító nem készít lokális másolatot a veremben, emiatt sokkal hatékonyabb kód jön létre tömb, rekord, string paraméterek használata esetén. A konstans formális paraméter helyén tetszőleges aktuális

paraméter (kifejezés, változó, érték) megadható. A **var** paraméterekhaz hasonlóan használható típus nélküli paraméter deklarálására is. Ebben az esetben azonban az aktuális paraméter csak változó lehet.

```
procedure ConstPar(const a,b: integer; var c: integer);
begin
  c := a + b;
  a := b; { hibás utasítás}
end;
```

Az Object Pascal nyelv a *nyitott* (*open*) paraméterek használatát is támogatja. Ilyen paraméterek segítségével az eljárások vagy függvények tetszőleges méretű tömbbel illetve stringgel hívhatók meg.

Ha a *string* típus helyett *OpenString* típust adunk meg, akkor az aktuális paraméter tetszőleges *string* típusú változó lehet. Az eljáráson vagy a függvényen belül a formális paraméter hossza mindig meg fog egyezni az aktuális string paraméter hosszával.

A *nyitott tömb* paraméterek lehetőséget biztosítanak arra, hogy olyan függvényeket vagy eljárásokat írjunk, melyek hívása nem függ a paramétertömb méretétől. Ehhez a formális paramétertömböt az *array of ElemTípus* deklarációval kell, hogy bevezessük. Az eljáráson vagy a függvényen belül úgy használhatjuk a nyitott tömböt, mintha *array[0..n-1] of ElemTípus* típusú lenne, ahol *n* az aktuális paraméterként átadott tömb elemeinek a száma. Az alprogramon belül a *High* függvénnyel kérdezhetjük le a nyitott tömb paraméter utolsó elemének az indexét. A *Low* függvény *0* értéket ad vissza és a *SizeOf* az aktuális paramétertőmb byteokban kifejezett méretével tér vissza.

```
function Media(a: array of integer): real;
var
  s: longint;
  i: integer;
begin
  s := 0;
  for i := 0 to High(a) do s := s + a[ i];
  Media := s / (High(a) + 1);
end;
```

Az Object Pascal érdekes lehetősége, hogy ha nyitott tömb formális paraméterrel deklaráltunk egy eljárást vagy egy függvényt, akkar az alprogram olyan konstans tömbbel is meghívható, amelynek az elemeit szögletes zárójel közé soroljuk fel.

```
writeln(Media([ 10,9,8,10,9,9,9,10,8,7,10] ));
```

megadhatunk típus nélküli nyitott tömb paramétereket is az *array of const* deklarációval. A Delphi ezt úgy oldja meg, hogy a *System* unitban egy *TVarRec* típust és hozzá tartozó konstansokat deklarál:

```
const
  vtInteger  = 0;
  vtBoolean  = 1;
  vtChar     = 2;
  vtExtended = 3;
  vtString   = 4;
  vtPointer  = 5;
  vtPChar    = 6;
  vtObject   = 7;
  vtClass    = 8;
type
  TVarRec = record
    case Integer of
        vtInteger: (VInteger: Longint; VType: Byte);
```

```
        vtBoolean: (VBoolean: Boolean);
        vtChar: (VChar: Char);
        vtExtended: (VExtended: PExtended);
        vtString: (VString: PString);
        vtPointer: (VPointer: Pointer);
        vtPChar: (VPChar: PChar);
        vtObject: (VObject: TObject);
        vtClass: (VClass: TClass);
end;
```

Az *array of const* deklaráció tulajdonképpen az *array of TVarRec* deklarációval azonos. A declarált *vtXXX* konstansok segítségével a tömb minden elemének a típusa lekérdezhető. A következő függvény egy tetszőleges tömb elemeit stringgé konkatenálja:

```
function MakeString(a: array of const): string;
var
  i: integer;
  s: string;
begin
s :='';
for i := 0 to High(a) do
  with a[ i] do
    case VType of
        vtInteger: s := s + IntToStr(VInteger);
        vtBoolean: case VBoolean of
              false: s := s + ' false';
              true: s := s + ' true';
              end;
        vtChar: s := s + VChar;
        vtExtended: s := s + FloatToStr(VExtended^);
        vtString: s := s + VString^;
     end;
  MakeString := s;
end;
```

Összetett típusú függvényértékek

A Pascalban megszokott típusok mellett az Object Pascal függvények ***összetett típusú*** (rekord, tömb, halmaz, stb.) értékekket is szolgáltathatnak vissza. Továbbra sem használhatók azonban az **object** és az állománytípusok.

Az *Assigned* függvény

Az előredefiniált Delphi függvények nagy része pointerekkel végez műveletet és *nil* értékkel tér vissza hiba esetén. Ezért a visszaadott értéket mindig tesztelni kell. Az *if p = nil* tesztet az Object Pascal hatékonyabbá és olvashatóbbá teszi az *if not Assigned(p)* teszt segítségével. Az *Assigned* függvény deklarációja tehát: *function Assigned(var p): boolean;*.

A null-terminál stringek használata

Az Object Pascal nyelv lehetőséget nyújt a C-ben megszokott null-terminál stringek használatára is. Ezeket a *PChar = ^Char*; vagy az *array[0..n] of char*; típusokkal lehet deklarálni. A stringeket egy *#0* karakter zárja le. A null-terminál stringeket extended (kibővített) szintaxis mellett lehet használni. Ezért a program a *{$X+}* direktívát kell, hogy tartalmazza. A null-terminál stringek használatát hatékony *SysUtils* unitbeli assembly rutinok segítik elő. Ilyen függvények pl.:

```
function StrLen(Str: PChar): word;          egy null-terminál string
                                                hosszát adja meg,
function StrUpper(Str: PChar: ): Pchar;     nagybetűssé alakít egy
                                                stringet,
```

```
function StrLower(Str: PChar): Pchar;      kisbetűssé alakít egy
                                               stringet,
function StrPas(Str: PChar): string;       null-terminál stringet
                                           Pascal típusúvá alakít,
function StrComp(s1, s2: PChar): integer; összehasonlít két
                                               stringet, stb.
```

Objektumok az Object Pascal nyelvben

Ahhoz, hogy tervezési időben is hozzá tudjunk férni az objektumokhoz, megfelelően ki kellett bővíteni a Delphi objektumorientált részét. A régi programokkal való kompatibilitás miatt természetesen megmaradt és ugyanúgy használható az *object* típus, de az új fogalomnak megfelelő fejlesztés csak a *class* típus és így az új alapelem, a komponens bevezetésével valósulhatott meg.

Az Object Pascalban bevezetett *class* típus felépítésében és használatában is hasonlít az *object* típusra. Lényeges különbség az, hogy a **class** típus példányai dinamikusan jönnek létre és minden új típusnak van elődje, a *TObject* típus.

Az adtatmezők és a metódusok mellett a *class* típusok jellemzőket (*property*) is tartalmazhatnak. A jellemző olyan névvel ellátott attribútuma a típusnak, amelyre csak az olvasás és/vagy az írás műveletét definiáljuk. Ebben a kontextusban elmondhatjuk tehát, hogy a jellemző definiciója az osztályban egy nevet, egy típust és műveleteket tartalmaz. A jellemzők képezik tulajdonképpen a Delphi által támogatott komponens-orientált fejlesztés alapját. Ezek a jellemzők mind tervezési, mind futási időben elérhetők.

```
property prop: integer read GetProp write SetProp;
```

ha a jellemző kifejezésben szerepel, akkor annak értékét a *read* direktíva után megadott adat vagy metódus szolgáltatja. Ha a jellemző értékadásban szerepel, akkor a megadott érték a *write* direktíva után megadott adatnak vagy metódusnak adódik át.

A jellemzők lehetséges típusai:

- egyszerű típus (numerikus, karakter, string)
- felsorolt típus
- halmaz
- objektum (a *TPersistent* típusból származtatott)
- tömb

A nem tömb jellemző definíciója más, opcionális tárolási direktívákat (***stored, default, nodefault***) is tartalmazhat a *read* és a *write* után.

```
property prop: integer read GetProp write SetProp stored true
                                                   default 10;
```

A *stored* direktívával azt jelzi a rendszer, hogy a jellemző értéke állományban írodott-e vagy sem. Így a *stored* direktíva lehetséges értékei: *true* vagy *false*, egy logikai típusú adat, egy logikai típusú értékkel visszatérő függvény (metódus). A *default* direktívával megadhatjuk a jellemző alapértelmezett értékét, illetve ha nincs ilyen, akkor ezt a *nodefault*-tal jelölhetjük.

Az adatrejtés új lehetőségei

Az objektum Pascalban megszokott belső (*private*) és kívülről is elérhető (*public*) adatai, metódusai és jellemzői mellett az Object Pascal még két adatrejtési módot definiál: a *protected* és a *published* elérhetőséget.

A *protected* (védett) elérhetőségű részei az objektumnak *private* elérésű a külvilág számára, ha azonban saját osztályt származtatunk a védett elemekkel rendelkező típusból, akkor ezek *public* elérésűvé válnak. A *published* direktíva ugyanúgy viselkedik, mint a *public*, azzal a különbséggel, hogy ezekhez az

adatmezőkhöz, jellemzőkhöz a rendszer futásidejű típusinformációkat kapcsol. A Delphi környezetben az *Object Inspector*nak van szüksége ilyen információkra.

Objektumpéldányok

A *class* típus valójában egy mutatótípus, amellyel létrehozott változó az objektumpéldányra fog mutatni. Az objektumpéldány számára memóriaterületet konstruktorral foglalunk, míg a terület felszabadításáról a destruktor gondoskodik. A *class* típus objektumainak a konstruktora a *Create*, destruktora pedig a *Destroy*. Ezekre az objektumokra ne használjuk a *new*és a *dispose* függvényeket, sem a ^ referenciát.

Objektumok hierarchiája

Az Object Pascal megtartotta a Pascal nyelv egyszeres öröklődését, tehát a *class* típusoknak is egyetlen közvetlen ősük lehet. Az előredefiniált *TObject* osztály minden osztály közös őse. Ha a típusdeklarációban elhagyjuk az ős osztály megadását, akkor automatikusan a *TObject* osztálytól fog származni az új típus. A *TObjectet* a *System* unit deklarálja:

```
type
  TObject = class;
  TClass = class of TObject;
  TObject = class
        constructor Create;
        destructor Destroy; virtual;
        procedure Free;
        class function NewInstance: TObject; virtual;
        procedure FreeInstance; virtual;
        class procedure InitInstance(Instance: Pointer): TObject;
        function ClassType: TClass;
        class function ClassName: string;
        class function ClassParent: TClass;
        class function ClassInfo: Pointer;
        class function InstanceSize: Word;
        class function InheritsForm(AClass: TClass): Boolean;
        procedure DefaultHandler(var Message); virtual;
        procedure Dispatch(var Message);
        class function MethodAddress(const Name: string): Pointer;
        class function MethodName(Address: Pointer): string;
        function FieldAddress(const Name: string): Pointer;
  end;
```

Az öröklődés során, a közvetlen ős osztály metódusainak elérése egyszerűbbé tehető az *inherited* direktíva felhasználásával.

Az Object Pascalban a polimorfizmus fogalma is kibővül. Most ez virtuális (*virtual*) és dinamikus (*dynamic*) metódusokon keresztül valósul meg. A konstruktoron kívül tetszőleges metódus virtuálissá tehető a *virtual*, illetve dinamikussá tehető a *dynamic* direktíva megadásával. Valamely medódus a hierarchia tetszőleges pontján virtuálissá vagy dinamikussá tehető, azonban a származtatott osztályoknál, ha ezeket a metódusokat újradefiniáljuk, az *override* direktívát kell, hogy használjuk. Pascal nyelvből tudjuk, hogy virtuális metódusok használatakor a fordítóprogram Virtuális Metódus Táblának (VMT) nevezett információs táblázatot készít, az osztályhoz kapcsolódóan. A dinamikus metódusok hívása úgy valósul meg, hogy a dinamikus metódusok adattáblázatai láncot alkotva, csak az adott osztályban definiált dinamikus metódusokról tárolnak információt. A meghívandó dinamikus metódus belépési címét egy rendszerrutin keresi meg a láncban.

Absztrakt metódusok

A hierarchiák többsége olyan típusokból (alaposztályokból) indul ki, amelynek virtuális, dinamikus metódusai csak arra szolgálnak, hogy teljessé tegyék a típus deklarációját. Ezek ún. absztrakt metódusokat tartalmaznak, amelyek az alaposztályban csak névlegesen vannak jelen, a származtatott osztályokban pedig mindenképpen újra kell őket definiálnunk. Egy metódus absztrakttá az *abstract* direktívával tehető:

```
procedure proc; virtual; abstract;
procedure proc; dynamic; abstract;
```

Osztályoperátorok

Az Object Pascal nyelv két olyan operátort definiál, amelyeknek operandusai osztály- illetve objektumhivatkozások. Az **is** operátort dinamikus típusellenőrzésre használjuk. Segítségével megtudhatjuk, hogy egy objektum az adott osztályhoz tartozik-e vagy sem:

```
object1 is ClassType
```

Az **as** operátort típuskonverzió végrehajtására használjuk:

```
object1 as ClassType
```

Osztálymetódusok

Az Object Pascal nyelv lehetőséget biztosít olyan metódusok létrehozására is, amelyek az objektumpéldány helyett magán az osztályon fejtik ki hatásukat. Ezek az ***osztálymetódusok*** (class methods). Ilyen metódusból természetesen sem az adatokat, sem a jellemzőket nem érhetjük el, hiszen ezek csak az objektum példányaiban léteznek. Egy medódust osztálymetódussá tudunk tenni úgy, hogy deklarációját a *class* direktívával vezetjük be:

```
type
 TMyObject = class
      class function GetName: string;
 end;

 class function TMyObject.GetName;
 begin
      GetName := 'TMyObject' ;
 end;

 begin
      writeln(TMyObject.GetName);
 end.
```

Üzenetkezelés

A Windows-filozófia alapja az üzenetkezelés. A Delphi üzenetkezelő-metódusok lehetővé teszik, hogy mi fogadjunk és megválaszoljunk dinamikusan továbbított üzeneteket. Egy ilyen metódust a *message* direktívával kell deklarálni. Az üzenetek azonítására és kezelésére a Windows rendelkezésünkre bocsájtja a *WM_XXX* konstansokat:

```
type
  TInputLine = class(TEdit)
      procedure WmKeyUp(var Message); message WM_KEYUP;
  end;
```

A Windows az üzeneteket továbítja, szórja. Üzenetet küldhetünk az objektumok Dispatch metódusával:

```
procedure TObject.Dispatch(var Message);
```

Kivételek kezelése

A kivételek (exception) olyan hibás események, amelyek megszakítják az alkalmazás szabályszerű futását. Ilyenkor a vezérlés a kivételkezelőnek adódik át. Az Object Pascal nyelv számos olyan eszközt tartalmaz, amelyek lehetővé teszik a kivételek, hibás események megkülönböztetését, kezelését. A kivételkezeléshez a Delphi saját objektumhierarchiát deklarál a *SysUtils* unitban. Ha ezt a unitot használjuk, a futás alatti (run-time) hibák automatikusan kivételekké alakulnak. Így olyan hibákat is ki lehet védni, mint például a memória túlcsordulás, általános védelmi hiba stb.

Az Object Pascalban a kivétel egyszerűen egy osztályként (*class*) van deklarálva. Az új kivételeket az *Exception* osztályból kell származtatni.

```
type
     EMathError = class(Exception);
```

Kivételeket a **raise** utasítás segítségével válthatunk ki:

```
raise[ objektumpéldány] [ at cím] ];
```

A következő eljárás ellenőrzi, hogy a beolvasott szám a [0..255] intervallumban van-e, ha nincs, akkor egy kivételt vált ki:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
   with Edit1 do
    if (StrToInt(Text) < 0) or (StrToInt(Text) > 255) then
         raise ERangeError.CreateFmt(' %d nincs a[ 0..255]
               intervallumban!' ,[ StrToInt(Text)] );
end;
```

Megfigyelhetjük, hogy a *raise* számára az objektumpéldányt közvetlenül az argumentumban hoztuk létre az *ERangeError* kivételosztály *CreateFmt* kontsruktorával. Fontos megjegyezni azt, hogy miután a kivételkezelés megtörtént, az objektumpéldány automatikus törlődik, vagyis a *Destroy* destruktor automatikusan meghívódik.

A kivételek kezelése a *try...except* utasítás segítségével történik:

```
try
     utasítások
except
     kivételek
[ else
     kivételek]
end;
```

A program végrehajtja a *try* utáni utasításokat, ha valamilyen kivétel lép fel, akkor a vezérlés ahhoz a legbelső kivételkezelőhöz kerül, amely alkalmas az adott osztályú kivételek kezelésére. Ha a blokk nem tartalmaz ilyen kivételkezelőt, akkor a program futása hibajelzéssel leáll. Ha az utasítás tartalmaz *else* részt is, akkor sikertelen keresés esetén ebben a részben leírtak fognak végrehajtódni.

A megfelelő kivételkezelő leírása az *on...do* utasítások segítségével történik. Ezeket a kivételkezelőket a beírás sorrendjében ellenőrzi a rendszer.

```
on[ azonosító:] típus do utasítás;
```

Az *azonosító* nem kötelező rész, ezt az utasítás végrehajtása során a kivételobjektum azonosítására használhatjuk.

Amikor az eljárás vagy függvény nem kezeli le a benne fellépő kivételt, továbbítani kell ezt az eljárást vagy a függvényt hívó külső programrésznek, vagyis újra elő kell idézni a kivételt. A kivételek ismételt előidézése a *raise* utasításnak az *except* részben történő megadásával végezhető el.

```
try
...
except
  on EOverflow do HandleOverflow;
  on EMathError do HandleMathError;
  on E: Exception do ErrorDialog(E.Message,E.HelpContext);
  raise;
else
  HandleOthers;
end;
```

Ha egy kódrész valamilyen erőforrást használ, mind normális, mind kivétellel megszakított esetben fell kell szabadítani a lefoglalt erőforrást. Ilyen esetekben a *try...finally* utasítást kell használni:

```
try
  utasítások;
finally
  utasítások;
end;
```

A program végrehajtja a *try* után következő utasításokat, ha valamilyen kivétel lép fel időközben, a vezérlés átadódik a *finally* résznek, ha nem lép fel kivétel, akkor is végrehajtódnak a *finally* részben leírtak. A *finally* rész utasításainak végrehajtása után a kivétel, ha volt ilyen, automatikusan ismét fellép, amelynek a kezelését általában egy külső *try...except* utasítás végzi el. Az utasítás feltétlenül szükséges olyan esetek kezelésére, amikor függetlenül a hibás vagy helyes végrehajtástól, bizonyos utasításokat végre kell hajtani. Például egy állományt a feldolgozása után mindig be kell zárni, bárhogy is fejeződött be ez a feldolgozás.

```
...
Assign(f,nev);
Reset(f);
try
  ProcessFile(f);
finally
  Close(f);
end;
```

Ha a *try* részben *Exit, Break* vagy *Continue* eljárást használunk, akkor a vezérlés átadódik a *finally* résznek. Ezek az eljárások használhatóak a kivételkezelőkből való kilépésre is, ekkor a kivétel automatikusan megszűnik.

Kovács Lehel

Tudománytörténet

Kémiatörténeti évfordulók

1997. szeptember–október

410 éve, 1587. október 21, 22, vagy 28-án született a németországi Lübeckben JOACHIM JUNGIUS, a „német Bacon". Az atomelmélet felújításában Boyle előfutára volt. A kísérletezés fontosságát hirdette. Kritizálta az alkímiát. Felismerte a levegő szerepét az égésben és azt tanította, hogy a réz kiválása rézgálic-oldatból vas hatására nem elemátalakulás, hanem egyenlő számú atom kicserélődése. 1657-ben halt meg.