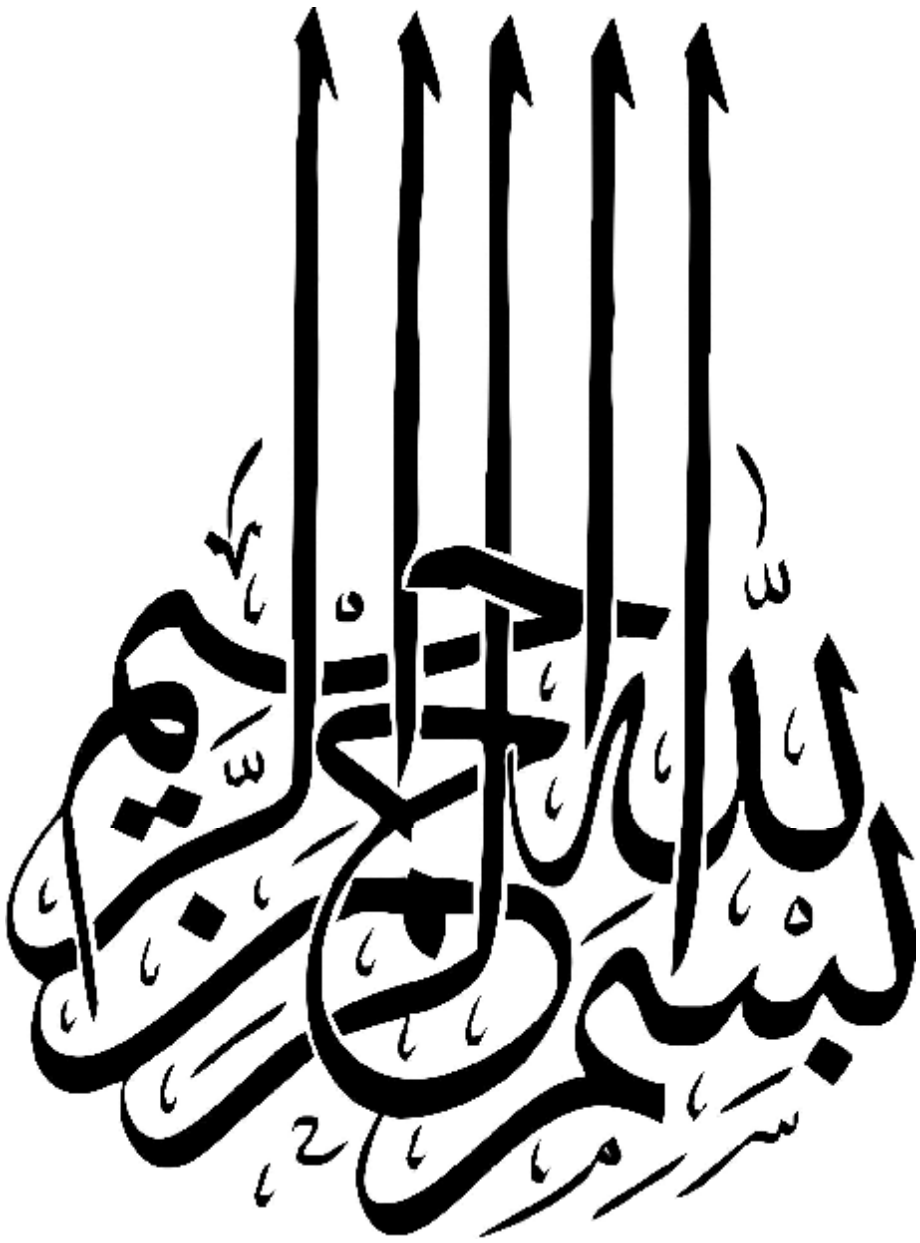

بسم الله الرحمن الرحيم

Object Pascal
Language Guide

راهنمای زبان پاسکال شیئی

مترجم : مهدی محبیان

Borland ®
Object Pascal

فصل ۱

سرآغاز

این کتاب زبان برنامه نویسی پاسکال شیئی را به همان صورتی که در ابزار توسعه بورلند به کار برده می‌شود، توصیف می‌کند.

آن چه که در این کتاب یافت می‌شود

هفت فصل اول کتاب، عناصر به کار رفته در برنامه نویسی شیء گرا را توضیح می‌دهند. فصل ۸ روال‌های استاندارد برای I/O فایل‌ها و دستکاری رشته‌ها را جمع‌بندی می‌کند.

فصول بعدی ملحقات و محدودیت‌های زبان را برای کتابخانه‌های اتصال پویا[۱] و بسته‌ها[۲] (فصل ۹)، و واسط‌های شیء(فصل ۱۰) شرح می‌دهند.

سه فصل پایانی عناوین پیشرفته زیر را نشان می‌دهند :

مدیریت حافظه (فصل ۱۱)، کنترل برنامه (فصل ۱۲)، و روال‌های زبان اسمبلی درمیان برنامه‌های پاسکال شیئی.

---

[۱] Dynamic-link library

[۲] Packages

استفاده از پاسکال شیئی

راهنمای زبان پاسکال شیئی برای توضیح دادن زبان پاسکال شیء گرا به منظور استفاده بر روی سیستم‌های عامل ویندوز یا لینوکس نوشته شده است. اختلاف‌های زبانی برای هر دو پلت فرم بسته به نیازمندی‌های پلت فرم، هرجایی که لازم باشد، مورد ملاحظه قرار می‌گیرند.

اغلب طراحان برنامه دلفی/کایلیکس کدهای پاسکال شیئی خود را در محیط توسعه یکپارچه (IDE)[۱] نوشته و کامپایل می‌کنند. کار در IDE به محصول اجازه اداره کردن بسیاری ازجزییات تنظیم پروژه‌ها و فایل‌های منابع، مانند نگهداری وابستگی اطلاعات در میان یونیت‌ها را می‌دهد. ممکن است که محصولات بورلند بر قیودی از سازماندهی برنامه تأکید کنند که به صورت أکید از آنها به عنوان ویژگی‌های زبان پاسکال شیء گرا صحبت نشده است. برای مثال، اگر شما برنامه‌ها را خارج از IDE نوشته و آنها را از طریق خط فرمان کامپایل کنید، می‌توان از قراردادهای خاص نام گذاری فایل و برنامه اجتناب کرد.

در این کتاب به طور کلی فرض می‌شود که شما در حال کار کردن در IDE هستید و برنامه‌های خود را با استفاده از کتابخانه اجزای ویژوال (VCL)[۲] بنا می‌کنید و/یا از کتابخانه اجزا بورلند برای پایگاه متقاطع (CLX)[۳] استفاده می‌کنید. اگرچه، برخی اوقات قوانین مخصوص بورلند از قوانینی که به تمامیت برنامه نویسی پاسکال شیئی اعمال می‌شوند، متمایز می‌شوند.

قراردادهای نگارشی

شناسه‌ها— یعنی اسامی ثابت‌ها، متغیرها، نوعها، فیلدها، خاصیت‌ها، روال‌ها، توابع، برنامه‌ها، یونیت‌ها، کتابخانه‌ها و بسته‌ها — به صورت *ایتالیک* در متن ظاهر می‌شوند. عملگرهای پاسکال شیئی، کلمات رزرو شده و دستور العمل‌ها به صورت **حروف درشت** خواهند بود. مثال‌ها برای کدنویسی و متونی که به صورت لفظ به لفظ به میان یک فایل یا درخط فرمان تایپ خواهید کرد به صورت حروف هم عرض و در یک پس زمینه خاکستری خواهند بود.

---

[۱] Integrated Development Environment

[۲] Visual Component Library

[۳] Borland Component Library for Cross Platform

در لیست برنامه‌های نشان داده شده، کلمات کلیدی و دستورالعمل‌ها به صورت **حروف درشت** ظاهر می‌شوند، درست به همان صورتی که در متن ظاهر خواهند شد:

```
function Calculate(X, Y: Integer): Integer;
begin
...
end;
```

چنان چه گزینه نشان دار کردن گرامر را به جریان انداخته باشید، این روش درست همان شیوه‌ای است که ویرایشگر کد کلمات رزرو شده و دستورالعمل‌ها را نمایش می‌دهد.

برخی لیست‌های برنامه، مانند مثال بالا، شامل علایم بریده گویی (مانند ...) هستند. بریده‌گویی‌ها بیانگر کد اضافی هستند که یک فایل واقعی شامل آنها خواهد بود. این حرف به معنای این است که آنها به طور لفظ به لفظ کپی نخواهند شد.

در توصیفات ترکیب نوشتاری/نحوی[۱]، *ایتالیک*‌ها، یعنی متونی که به صورت کج نوشته خواهند شد، بیانگر نگهدارنده مکان برای جایی هستند که در کد واقعی آنها را با ساختارهای معتبری از نظر قواعد جایگزین خواهید کرد. برای مثال، هدینگ اعلان تابع می‌تواند مانند زیر بیان شود:

```
function functionName(argumentList): returnType;
```

همچنین توصیفات گرامر و ترکیب نوشتاری/نحوی می‌توانند شامل علایم بریده گویی (...) و زیرنگانشت‌ها باشند:

```
function functionName(arg1, ..., argn): ReturnType;
```

---

۱ Syntax

۲

فصل

مقدمه

پاسکال شیئی یک زبان سطح بالا، کامپایل شده و به طور قدرتمندی نوع دار است که طراحی ساخت‌یافته و شیءگرا را پشتیبانی می‌کند. مزایای آن شامل خوانش آسان کد، ترجمه و کامپایل سریع و استفاده از یونیت‌های متعدد برای برنامه نویسی پیمانه ای[۱] (مدولار) می‌باشد.

پاسکال شیئی مشخصه‌های ویژه ای دارد که از چارچوب اجزای بورلند و محیط RDA پشتیبانی می‌کنند. اکثراً، برای توضیحات و مثال‌ها در این کتاب، فرض براین است که با بهره گیری از ابزار توسعه بورلند مانند دلفی یا کایلیکس، از پاسکال شیئی برای طراحی برنامه‌ها استفاده می‌کنید.

مدیریت برنامه

برنامه‌ها معمولاً به واحدهای (مدول‌ها) کد منبع که یونیت خوانده می‌شوند، تقسیم می‌شوند. هر برنامه با یک هدینگ که یک نام برای برنامه تعیین می‌کند، شروع می‌شود. هدینگ یا سرفصل با یک شرط **uses** اختیاری پی گرفته می‌شود. سپس بلوکی از اعلان‌ها و دستورها می‌آید. شرط **uses** یونیت‌هایی

---

[۱] Modular programming

را لیست می‌کند که به برنامه پیوند شده اند، این یونیت‌ها—که می‌توانند بوسیله برنامه‌های مختلف به اشتراک گذاشته شوند—اغلب شرط‌های **uses** مال خود را دارا هستند.

شرط **uses** برای کامپایلر اطلاعاتی درباره وابستگی‌ها در میان واحدها (مدول‌ها) تدارک می‌بینند. از آن جایی که این اطلاعات در خود واحدها (مدول‌ها) ذخیره شده اند، برنامه‌های پاسکال شیئی نیاز به فایل‌های ساخت (makefiles)، پرونده‌های هدر (header files)، یا دستور پیش پردازنده "include" ندارند. (مدیر پروژه هر بار که پروژه در IDE بارگذاری می‌شود، یک فایل ساخت –makefile– تولید می‌کند، اما صرفاً این فایل‌ها را برای گروه‌های پروژه‌ای که بیشتر از یک پروژه را شامل می‌شوند، ذخیره می‌کند.) برای بحث بیشتر درباره ساختار برنامه و وابستگی‌ها، بخش «برنامه‌ها و یونیت‌ها» را در فصل۳ ملاحظه نمایید.

فایل‌های منبع پاسکال

کامپایلر انتظار دارد که کد منبع پاسکال را در سه نوع فایل پیدا کند:

- فایل‌های منبع *unit* (که با پسوند pas. پایان می‌یابند)
- فایل‌های *project* (که با پسوند dpr. خاتمه می‌یابند)
- فایل‌های منبع *package* (که با پسوند dpk. خاتمه می‌یابند)

فایل‌های منبع یونیت (Unit) بیشترین کد یک برنامه را در خود جای می‌دهند. هر برنامه یک فایل منفرد پروژه و چندین فایل یونیت دارد؛ فایل پروژه —که با فایل برنامه "main" در پاسکال سنتی متناظر است— فایل‌های یونیت را در یک برنامه سازماندهی می‌کند. ابزار توسعه بورلند به طور خودکار یک فایل پروژه را برای هر برنامه نگهداری کرده و پشتیبانی می‌کند.

اگر یک برنامه را از طریق خط فرمان کامپایل کنید، می‌توانید همه کدهای منبع خود را در فایل‌های یونیت (pas.) قرار دهید. اما درصورتی که از IDE برای ساختن برنامه خود استفاده می‌کنید، بایستی یک فایل پروژه (dpr.) داشته باشید.

فایل‌های منبع Package شبیه فایل‌های پروژه هستند، اما آنها برای ساختن کتابخانه‌های قابل پیوند به صورت پویا به نام *package* مورد استفاده قرار می‌گیرند. برای اطلاعات بیشتر درباره بسته‌ها، بخش «کتابخانه‌ها و بسته‌ها» را در فصل ۹ ملاحظه نمایید.

دیگرفایل‌های به کار رفته برای ایجاد برنامه‌ها

علاوه بر واحدهای (یا مدول‌های) کد منبع، محصولات بورلند از فایل‌های غیرپاسکال گوناگونی برای ساخت برنامه‌ها استفاده می‌کنند. این فایل‌ها که به طور خودکار نگهداری و پشتیبانی می‌شوند، شامل موارد زیر هستند:

- فایل‌های *form* که با پسوند dfm. (دلفی) یا xfm. (کایلیکس) خاتمه می‌یابند.
- فایل‌های *resource* که با پسوند res. خاتمه می‌یابند و
- فایل‌های *project options* که با پسوند dof. (دلفی) یا kof. (کایلیکس) خاتمه می‌یابند.

یک فایل فرم یا یک فایل متنی یا یک فایل منبع کامپایل شده است که می‌تواند شامل بیت مپ‌ها، رشته‌ها و ... باشد. هر فایل فرم یک فرم یکتا را نمایش می‌دهد که معمولاً با یک پنجره یا یک جعبه محاوره‌ای واقع در برنامه متناظر است.

IDE به شما اجازه می‌دهد تا فایل‌های فرم را به صورت متن مشاهده و ویرایش کنید و فایل‌های فرم را یا به صورت متنی یا باینری ذخیره نمایید. گرچه رفتار پیش فرض، ذخیره فایل‌های فرم به صورت متنی است. آن‌ها معمولاً به طور دستی ویرایش نمی‌شوند؛ استفاده از ابزار طراحی ویژوال بورلند به منظور ویرایش فایل‌های فرم امر متداولی است. هر پروژه حداقل یک فرم دارد و هر فرم یک فایل یونیت (pas.) متناظر دارد که به طور پیش فرض، همان اسمی را که فایل فرم دارد، دارا می‌باشد.

علاوه بر فایل‌های فرم، هر پروژه از یک فایل منبع (res.) برای نگه داری بیت مپ مربوط به آیکون برنامه، استفاده می‌کند. به طور پیش فرض این فایل همان اسمی را دارد که فایل پروژه (dpr.) دارد. برای تغییر آیکون یک برنامه، از Project Options استفاده کنید. یک فایل گزینه‌های پروژه (dof. یا kof.) شامل تنظیمات کامپایلر و پیوند دهنده، دایرکتوری‌های جستجو، اطلاعات ورژن و ... می‌باشد. هر پروژه یک فایل گزینه‌های پروژه وابسته با همان نام فایل پروژه (dpr.) دارد. معمولاً، گزینه‌ها در این فایل از طریق Project Options تنظیم می‌شوند.

ابزارهای متعدد در IDE ، داده‌ها را در انواع دیگری از فایل‌ها ذخیره می‌کنند. فایل‌های تنظیمات میزکار (dsk. یا desk.) محتوی اطلاعاتی درباره آرایش پنجره‌ها و دیگر گزینه‌های پیکربندی هستند؛ تنظیمات میز کار می‌تواند خاص پروژه یا برای کلیت محیط باشد. این فایل‌ها اثر مستقیم بر کامپایل ندارند.

فایل‌های تولید شده توسط کامپایلر

اولین بار که یک برنامه یا یک کتابخانه اتصال پویای استاندارد را ایجاد می‌کنید، کامپایلر یک فایل یونیت کامپایل شده dcu. (برای ویندوز) یا dcu/.dpu. (برای لینوکس) برای هر یونیت تازه به کار رفته در پروژه شما، تولید می‌کند؛ سپس همه فایل‌های dcu. (برای ویندوز) dcu/.dpu. (برای لینوکس) واقع در پروژه شما به هم متصل می‌شوند تا یک فایل قابل اجرای واحد یا یک فایل کتابخانه اشتراکی یکتا را ایجاد کنند. اولین بار که شما یک بسته (package) را ایجاد می‌کنید، کامپایلر یک فایل dcu. (برای ویندوز) dcu/.dpu. (برای لینوکس) برای هر یونیت تازه جای گرفته در بسته (package) ایجاد می‌کند و سپس هم یک فایل dcp. و هم یک فایل بسته را ایجاد می‌کند. (برای اطلاعات بیشتر درباره کتابخانه‌ها و بسته‌ها، فصل۹ را ملاحظه نمایید.) در صورتی که از سوییچ **GD-** استفاده کنید، پیونددهنده یک فایل map و یک فایل drc. تولید می‌کند؛ فایل drc. که شامل منابع رشته می‌باشد، می‌تواند به یک فایل منبع کامپایل شود.

زمانی که یک پروژه را بازسازی می‌کنید، واحدهای مجزا مجدداً کامپایل نمی‌شوند مگر آنکه فایل‌های منبع (pas.) آنها از زمان آخرین کامپایل تغییر کرده باشند یا فایل‌های dcu. (برای ویندوز) یا dcu/.dpu. (برای لینوکس) آنها یافت نشود، یا شما صریحاً به کامپایلر بگویید که آنها را مجدداً پردازش کند. در واقع، مادامی که کامپایلر بتواند فایل یونیت کامپایل شده را پیدا کند، به هیچ وجه لازم نیست فایل منبع یک یونیت حضور داشته باشد.

برنامه‌های نمونه

مثال‌هایی که در پی می‌آیند، ویژگی‌های پایه‌ای برنامه نویسی پاسکال شیئی را نشان می‌دهند. مثال‌ها برنامه‌های ساده پاسکال شیئی را نمایش می‌دهند که نمی‌توانند از طریق IDE کامپایل شوند؛ اما شما می‌توانید آنها را از طریق خط فرمان کامپایل کنید.

یک برنامه کنسول ساده

برنامه‌ای که می‌آید، یک برنامه کنسول ساده است که می‌توانید آن را از خط فرمان کامپایل کرده و اجرا نمایید.

```
program Greeting;
{$APPTYPE CONSOLE}
var MyMessage: string;
begin
MyMessage := 'Hello world!';
Writeln(MyMessage);
end.
```

خط اول یک برنامه با نام *Greeting* را اعلان می‌کند. دستور {$APPTYPE CONSOLE} به کامپایلر می‌گوید که این برنامه یک برنامه کنسول است که از طریق خط فرمان اجرا خواهد شد. خط بعدی یک متغیر با نام *MyMessage* اعلان می‌کند که یک رشته را نگه می‌دارد. (پاسکال شیئی انواع داده رشته‌ای اصلی را داراست.) سپس برنامه عبارت "Hello world!" را به متغیر *MyMessage* تخصیص می‌دهد و محتوای *MyMessage* را با استفاده از روال *Writeln* به خروجی استاندارد ارسال می‌کند. (*Writeln* به طور ضمنی در یونیت *System* که کامپایلر به طور خودکار آن را ضمیمه هر برنامه می‌کند، تعریف شده است.)

شما می‌توانید این برنامه را در یک فایل با نام Greeting.pas یا Greeting.dpr تایپ کرده و آن را با وارد کردن یکی از دستورات زیر در خط فرمان کامپایل کنید:

```
در دلفی :  DCC32 Greeting
در کایلیکس :  dcc Greeting
```

فایل اجرایی حاصل پیغام "Hello world!" را چاپ می‌کند.

قطع نظر از سادگی، این مثال از چند نظر با برنامه‌هایی که شما احتمالاً با ابزارهای توسعه بورلند خواهید نوشت، تفاوت خواهد داشت. اولاً این یک برنامه کنسول است. ابزارهای توسعه بورلند نوعاً برای نوشتن برنامه‌ها با رابط‌های گرافیکی مورد استفاده قرار می‌گیرند؛ از این رو، شما معمولاً *Writeln* را فراخوانی نخواهید کرد. علاوه بر این، سرتاسر برنامه مثال زده شده (ذخیره شده برای *Writeln*) در یک فایل منفرد قرار دارد. در یک برنامه نوعی، هدینگ برنامه— خط اول مثال— در یک فایل مجزا قرار خواهد گرفت که غیر از اندکی فراخوانی به متدهای تعریف شده در فایل‌های یونیت، حاوی هیچ منطق واقعی برای برنامه نخواهد بود.

یک مثال پیچیده تر

مثال بعدی برنامه‌ای را نشان می‌دهد که به دو فایل تقسیم شده است: یک فایل پروژه و یک فایل یونیت. فایل پروژه، که می‌توانید آن را با نام Greeting.dpr ذخیره کنید به این صورت به نظر می‌رسد:

```
program Greeting;
{$APPTYPE CONSOLE}
uses Unit1;
begin
PrintMessage('Hello World!');
end.
```

خط اول یک برنامه با نام *Greeting* را، که بار دیگر، یک برنامه کنسول است اعلان می‌کند. عبارت **uses** Unit1; به کامپایلر می‌گوید که *Greeting* شامل یک یونیت به نام *Unit1* است. در نهایت، برنامه روال *PrintMessage* را با ارسال عبارت "Hello world!" به آن فرا می‌خواند. سؤال این است که روال *PrintMessage* از کجا آمده است؟ این روال در *Unit1* تعریف شده است. در زیر کد منبع مربوط به *Unit1* که می‌توانید آن را در فایلی به نام Unit1.pas ذخیره کنید، آمده است:

```
unit Unit1;
interface
procedure PrintMessage(msg: string);
implementation
procedure PrintMessage(msg: string);
begin
Writeln(msg);
end;
end.
```

Unit1 یک روال با نام *PrintMessage* تعریف می‌کند که یک رشته را به عنوان یک آرگومان گرفته و آن را به خروجی استاندارد ارسال می‌کند. (در پاسکال، روتین‌هایی که یک مقدار را برگشت ندهند روال[۱] نامیده می‌شوند. روتین‌هایی که یک مقدار را برگشت می‌دهند تابع[۲] خوانده می‌شوند.) توجه کنید که *PrintMessage* دو مرتبه در *Unit1* اعلان شده است. اعلان اول، زیر کلمه رزرو شده **interface**، *PrintMessage* را برای مدول‌های دیگر (مانند *Greeting*) که از *Unit1* استفاده می‌کنند، قابل دسترس می‌سازد. اعلان دوم، زیر کلمه رزرو شده **implementation**، عملاً *PrintMessage* را تعریف می‌کند.

اکنون می‌توانید *Greeting* را با وارد کردن یکی از دستورات زیر در خط فرمان، کامپایل کنید :

---

[۱] Procedure

[۲] Function

در دلفی : DCC32 Greeting

در کایلیکس : dcc Greeting

دراین جا دیگر لازم نیست که *Unit1* را به عنوان یک آرگومان در خط فرمان قرار دهید. زمانی که کامپایلر Greeting.dpr را پردازش می‌کند، به طور خودکار به دنبال فایل‌های یونیتی می‌گردد که برنامه *Greeting* به آنها وابسته است. فایل اجرایی حاصل همان نتیجه‌ای را دارد که مثال اولمان داشت: پیغام "Hello world!" را چاپ می‌کند.

یک برنامه اصلی

مثال بعدی ما برنامه‌ای است که با استفاده از اجزای VCL یا CLX در IDE ایجاد می‌شود. این برنامه از فایل‌های فرم و منبعی که به طور خودکار ایجاد شده اند، استفاده می‌کند، بنابراین شما صرفاً قادر به کامپایل آن از طریق کد منبع نخواهید بود. اما این برنامه ویژگی‌های مهمی از پاسکال شیئی را نمایش می‌دهد. علاوه بر یونیت‌های متعدد، برنامه از کلاس‌ها و اشیاء که در فصل۷ به طور مفصل بحث خواهند شد، استفاده می‌کند.

برنامه شامل یک فایل پروژه و دو فایل یونیت جدید است. نخست، فایل پروژه:

```
program Greeting; { comments are enclosed in braces }
uses
Forms, {change the unit name to QForms on Linux}
Unit1 in 'Unit1.pas' { the unit for Form1 },
Unit2 in 'Unit2.pas' { the unit for Form2 };
{$R *.res} { this directive links the project's resource file }
begin
{ calls to Application }
Application.Initialize;
Application.CreateForm(TForm1, Form1);
Application.CreateForm(TForm2, Form2);
Application.Run;
end.
```

بار دیگر، برنامه خود را *Greeting* می‌نامیم. برنامه از سه یونیت استفاده می‌کند: *Forms*، که بخشی از VCL و CLX است؛ *Unit1*، که متناظر با فرم اصلی برنامه (*Form1*) است؛ و *Unit2*، که متناظر با فرم دیگری می‌باشد (*Form2*).

برنامه یک سری فراخوانی به یک شیء به نام *Application* که وهله‌ای از کلاس *TApplication* تعریف شده در یونیت *Forms* است، انجام می‌دهد. (هر پروژه به طور خودکار یک شیء *Application* تولید

می‌کند.) دو تا از این فراخوانی‌ها یک متد *TApplication* با نام *CreateForm* را احضار می‌کنند. فراخوانی اول برای *CreateForm*، *Form1* را که یک وهله از کلاس *TForm1* تعریف شده در *Unit1* است، ایجاد می‌کند. فراخوانی دوم برای *CreateForm*، *Form2* را که یک وهله از کلاس *TForm2* تعریف شده در *Unit2* است، ایجاد می‌کند. *Unit1* مانند این به نظر می‌رسد:

```
unit Unit1;
interface
uses { these units are part of the Visual Component Library (VCL) }
Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs, StdCtrls;
{
On Linux, the uses clause looks like this:
uses { these units are part of CLX }
SysUtils, Types, Classes, QGraphics, QControls, QForms, QDialogs;
}
type
TForm1 = class(TForm)
Button1: TButton;
procedure Button1Click(Sender: TObject);
end;
var
Form1: TForm1;
implementation
uses Unit2; { this is where Form2 is defined }
{$R *.dfm} { this directive links Unit1's form file }
procedure TForm1.Button1Click(Sender: TObject);
begin
Form2.ShowModal;
end;
end.
```

Unit1 یک کلاس به نام *TForm1* (مشتق شده از *TForm*) و یک وهله از این کلاس، *Form1* را ایجاد می‌کند. *TForm1* شامل یک دکمه— *Button1*، یک وهله از *TButton*— و یک روال به نام *TForm1.Button1Click* است که این روال هنگام اجرای برنامه، زمانی که کاربر *Button1* را کلیک کند، فراخوانده می‌شود. *TForm1.Button1Click*، *Form1* را مخفی می‌کند و *Form2* را نمایش می‌دهد (فراخوانی *Form2.ShowModal*). *Form2* در *Unit2* تعریف شده است:

```
unit Unit2;
interface
uses
Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
StdCtrls;
{
On Linux, the uses clause looks like this:
uses { these units are part of CLX }
SysUtils, Types, Classes, QGraphics, QControls, QForms, QDialogs;
}
type
TForm2 = class(TForm)
Label1: TLabel;
CancelButton: TButton;
procedure CancelButtonClick(Sender: TObject);
```

```
procedure FormClose(Sender: TObject; var Action: TCloseAction);
end;
var
Form2: TForm2;
implementation
uses Unit1;
{$R *.dfm}
procedure TForm2.CancelButtonClick(Sender: TObject);
begin
Form2.Close;
end;
end.
```

Unit2 یک کلاس به نام *TForm2* و یک وهله از این کلاس، یعنی *Form2* را ایجاد می‌کند. *TForm2* شامل یک دکمه (*CancelButton* به عنوان وهله ای از *TButton*) و یک برچسب (*Label1* به عنوان یک وهله از *TLabel*) می‌باشد. شما نمی‌توانید کد منبع این فرم را ببینید، اما *Label1* یک عنوان را نمایش می‌دهد که "Hello world!" را می‌خواند. این عنوان در فایل فرم متعلق به *Form2* (Unit2.dfm) تعریف شده است.

Unit2 یک روال را تعریف می‌کند. درحین اجرا، زمانی که کاربر *CancelButton* را کلیک کند، *TForm2.CancelButtonClick* فراخوانده شده و *Form2* را می‌بندد. این روال (همراه با *TForm1.Button1Click* برای *Unit1*) به عنوان یک *گرداننده رویداد*[۱] شناخته می‌شود زیرا به رویدادهایی که در هنگام اجرای برنامه روی می‌دهند، عکس‌العمل نشان می‌دهد. توسط فایل‌های فرم (dfm. در ویندوز و xfm. در لینوکس) مربوط به *Form1* و *Form2* گرداننده‌های رویداد به رویدادهای معینی منتسب می‌شوند.

هنگامی که برنامه *Greeting* شروع می‌شود، *Form1* نمایش می‌یابد و *Form2* غیرقابل مشاهده است. (به طور پیش فرض تنها اولین فرم ایجاد شده در فایل پروژه در زمان اجرا قابل مشاهده است. این فرم به عنوان فرم اصلی پروژه، یعنی *main form* شناخته می‌شود.) زمانی که کاربر دکمه‌ای را که روی *Form1* قرار دارد کلیک کند، *Form1* ناپدید شده و *Form2* جای آن را می‌گیرد که خوشامدگویی "Hello world!" را نمایش می‌دهد. چنان چه کاربر (با کلیک کردن بر روی دکمه *CancelButton* یا دکمه **X** در میله عنوان) *Form2* را ببندد، *Form1* مجدداً ظاهر می‌شود.

---

[۱] event handler

فصل ۳

برنامه‌ها و یونیت‌ها

یک برنامه از واحدهای کد منبع به نام یونیت‌ها تشکیل شده است. هر یونیت[۱] در فایل مخصوص به خود ذخیره شده و به صورت مجزا کامپایل می‌شود؛ یونیت‌های کامپایل شده به یکدیگر متصل می‌شوند تا یک برنامه را ایجاد کنند. یونیت‌ها به شما اجازه می‌دهند که:

- برنامه‌های بزرگ را به واحدهایی تقسیم کنید که به صورت مجزا قابل ویرایش باشند.
- کتابخانه‌هایی (مجموعه برنامه‌ها) را ایجاد کنید که بتوانید درمیان برنامه‌ها به اشتراک بگذارید.
- برنامه‌ها را به منظور استفاده برنامه نویسان دیگر توزیع کنید بدون اینکه مجبور باشید کد منبع را در دسترس قرار دهید.

در برنامه نویسی پاسکال سنتی، تمامی کد منبع که در برنامه اصلی قرار دارد، در فایل‌های pas. ذخیره می‌شود. ابزار بورلند از یک فایل پروژه (dpr.) برای ذخیره برنامه «اصلی» استفاده می‌کنند، در حالی که اغلب کد منبع در فایل‌های یونیت (pas.) جای می‌گیرد. هر برنامه— یا پروژه— از یک فایل پروژه یکتا و یک یا چند فایل یونیت تشکیل می‌شود. (به عبارت دقیق‌تر، شما صریحاً نیاز ندارید که از هر یونیتی در یک پروژه استفاده کنید، اما همه برنامه‌ها به طور خودکار از یونیت *System* استفاده می‌کنند.)

---

[۱] Unit

برای بناکردن[۱] یک پروژه، کامپایلر یا به یک فایل منبع یا یک فایل یونیت کامپایل شده برای هر یونیت نیاز دارد.

ساختار و ترکیب نوشتاری/نحوی برنامه

یک برنامه شامل موارد زیر است:

- هدینگ برنامه
- یک شرط (اختیاری) **uses** و
- بلوکی از اعلان‌ها و دستورات.

هدینگ برنامه یک نام برای برنامه تعیین می‌کند. شرط اختیاری **uses** یونیت‌های استفاده شده توسط برنامه را لیست می‌کند. بلوک نیز شامل اعلان‌ها و دستوراتی است که هنگام راه‌اندازی برنامه، اجرا می‌شوند. IDE انتظار دارد که این سه عنصر را در فایل پروژه (dpr.) یکتایی پیدا کند.

مثال زیر فایل پروژه را برای یک برنامه با نام Editor نشان می‌دهد.

```
1  program Editor;
2
3  uses
4  Forms, {change to QForms in Linux}
5  REAbout in 'REAbout.pas' {AboutBox},
6  REMain in 'REMain.pas' {MainForm};
7
8  {$R *.res}
9
10 begin
11 Application.Title := 'Text Editor';
12 Application.CreateForm(TMainForm, MainForm);
13 Application.Run;
14 end.
```

سطر ۱ محتوی هدینگ برنامه است. شرط **uses** در سطور ۳ تا ۶ قرار دارد. سطر ۸ یک فرمان کامپایلر است که فایل منبع پروژه را به برنامه متصل می‌کند. سطور ۱۰ تا ۱۴ شامل دستوراتی است که هنگام شروع برنامه، اجرا می‌شوند. بالاخره، فایل پروژه، مانند همه فایل‌های منبع با یک نقطه پایان می‌یابد.

---

[۱] Build

در حقیقت این یک فایل پروژه صاف و ساده است. فایل‌های پروژه معمولاً کوچک هستند، زیرا که بیشتر منطق برنامه در فایل‌های یونیت پروژه جای می‌گیرند. فایل‌های پروژه به طور خودکار تولید و پشتیبانی می‌شوند و به ندرت لازم می‌شود که آنها را به صورت دستی ویرایش کرد.

هدینگ برنامه

هدینگ برنامه، نام برنامه را تعیین می‌کند. هدینگ شامل کلمه رزرو شده **program** است، که به دنبال آن یک شناسه معتبر می‌آید و بعد از آن هم یک نقطه ویرگول می‌آید. شناسه بایستی با نام پروژه مطابقت داشته باشد. در مثال بالا، از آن جایی که برنامه Editor نامیده شده، فایل پروژه بایستی EDITOR.dpr نامیده شود.

در پاسکال استاندارد، هدینگ یک برنامه می‌تواند شامل پارامترهایی بعد از نام پروژه باشد:

```
program Calc(input, output);
```

کامپایلر پاسکال شیئی بورلند از این پارامترها صرف نظر می‌کند.

شرط uses برنامه

شرط **uses** یونیت‌هایی را که در برنامه جای گرفته‌اند، لیست می‌کند. این یونیت‌ها ممکن است در صورت تمایل هرکدام شرط **uses** مخصوص به خود را دارا باشند. برای اطلاعات بیشتر درباره شرط **uses**، بخش «ارجاعات یونیت و شرط **uses**» را در همین فصل ببینید.

بلوک

بلوک شامل یک دستور ساده یا ساخت یافته است که هنگام شروع برنامه اجرا می‌شود. در اغلب برنامه‌ها، بلوک از دستورات مرکب و ترکیبی تشکیل می‌شود—که میان کلمات رزرو شده **begin** و **end** جای می‌گیرند—که دستورات اجزای آنها متدهای ساده‌ای هستند که به شیء *Application* پروژه فراخوانده می‌شوند. (هر پروژه یک متغیر *Application* دارد که یک وهله *TApplication*، *TWebApplication* یا *TServiceApplication* را نگه می‌دارد.) بلوک هم چنین می‌تواند شامل اعلان ثابت‌ها، نوع‌ها، متغیرها، روال‌ها و توابع باشد؛ این اعلان‌ها می‌بایست مقدم بر بخش دستور بلوک باشند.

ساختار و ترکیب یونیت

یک یونیت از نوع‌ها (از جمله کلاس‌ها)، ثابت‌ها، متغیرها و روتین‌ها (روال‌ها و توابع) تشکیل می‌شود. هر یونیت در فایل یونیت (pas.) مخصوص به خود تعریف می‌شود.

یک فایل یونیت با یک هدینگ یونیت شروع می‌شود که با بخش‌های *interface*، *implementation*، *initialization* و *finalization* پی گرفته می‌شود. بخش‌های *initialization* و *finalization* اختیاری هستند. استخوان‌بندی فایل یونیت به این صورت است:

```
unit Unit1;
interface
uses { List of units goes here }
{ Interface section goes here }
implementation
uses { List of units goes here }
{ Implementation section goes here }
initialization
{ Initialization section goes here }
finalization
{ Finalization section goes here }
end.
```

یونیت بایستی با کلمه **end** که پس از آن یک نقطه می‌آید، خاتمه یابد.

هدینگ یونیت

هدینگ یونیت، نام یونیت را تعیین می‌کند. هدینگ شامل کلمه رزرو شده **unit** است، که به دنبال آن یک شناسه معتبر می‌آید و بعد از آن هم یک نقطه ویرگول می‌آید. برای برنامه‌های طراحی شده با استفاده از ابزار بورلند، شناسه بایستی با نام فایل یونیت مطابقت داشته باشد. از این رو، هدینگ یونیت

```
unit MainForm;
```

در یک فایل منبع با نام MAINFORM.pas ظاهر خواهد شد و فایل محتوی یونیت کامپایل شده، MAINFORM.dcu خواهد بود.

نام یونیت‌ها در میان یک پروژه بایستی منحصر به فرد باشند. حتی اگر فایل‌های یونیت در دایرکتوری‌های جداگانه‌ای قرار داشته باشند، دو یونیت با نام یکسان نمی‌توانند در یک پروژه مورد استفاده قرار گیرند.

بخش واسط (interface)

بخش واسط یک یونیت با کلمه رزرو شده **interface** شروع می‌شود و تا زانی که بخش پیاده‌سازی (**implementation**) آغاز شود، ادامه می‌یابد. بخش واسط ثابت‌ها، نوع‌ها، متغیرها، روال‌ها و توابعی را که در دسترس مشتری‌ها هستند، اعلان می‌کند—که این مشتری‌ها یونیت‌ها و برنامه‌های دیگری هستند که از یونیت جایی که اعلان شده، استفاده می‌کنند. این موجودیت‌ها عمومی[۱] خوانده می‌شوند زیرا یک مشتری می‌تواند از آنها استفاده کند به طوری که انگار آنها برای خود مشتری اعلان شده‌اند.
اعلان واسط یک روال یا تابع تنها شامل هدینگ روتین است. بلوک روال یا تابع در بخش **implementation** می‌آید. بنابراین اعلان روال و تابع در بخش واسط مانند اعلان پیشرو (forward) عمل می‌کند، هر چند از دستور **forward** استفاده نشده باشد.

اعلان **interface** یک کلاس باید شامل اعلان برای همه اعضای کلاس باشد.

بخش واسط می‌تواند دارای شرط **uses** مربوط به خود باشد که آن هم بایستی فوراً بعد از کلمه **interface** بیاید. برای اطلاعات بیشتر درباره شرط **uses** بخش «ارجاعات یونیت و شرط **uses**» را در همین فصل ملاحظه نمایید.

بخش پیاده‌سازی (implementation)

بخش پیاده‌سازی (implementation) یک یونیت با کلمه رزرو شده **implementation** شروع شده و تا شروع بخش ارزش‌گذاری (initialization) ادامه می‌یابد یا، اگر بخش ارزش‌گذاری وجود نداشته باشد، تا انتهای یونیت ادامه می‌یابد. بخش پیاده‌سازی (implementation) روال‌ها و توابعی را که در بخش واسط (interface) اعلان شده اند، تعریف می‌کند. در میان بخش پیاده‌سازی، این روال‌ها و توابع ممکن است به هر ترتیبی تعریف شده و فراخوان شوند. زمانی که روال‌ها و توابع را در بخش پیاده‌سازی تعریف می‌کنید، می‌توانید از لیست پارامترها برای هدینگ روال‌ها و توابع عمومی صرف نظر کنید؛ اما اگر لیستی از پارامترها را جای دادید باید دقیقاً با بخش واسط مطابقت داشته باشد.

---

[۱] public

علاوه بر تعریف توابع و روال‌های عمومی، بخش پیاده‌سازی (implementation) می‌تواند ثابت‌ها، نوع‌ها (از جمله کلاس‌ها)، متغیرها، روال‌ها و توابعی را که برای یونیت خصوصی[۱] هستند، اعلان کند— خصوصی یعنی این که برای مشتری‌ها غیر قابل دسترس باشند.

بخش پیاده‌سازی می‌تواند دارای شرط **uses** مال خود باشد، که بایستی بلافاصله بعد از واژه **implementation** بیاید. برای اطلاعات بیشتر درباره شرط **uses** بخش «ارجاعات یونیت و شرط **uses**» را در همین فصل ملاحظه نمایید.

بخش ارزش گذاری (initialization)

بخش ارزش‌گذاری (initialization) اختیاری است. این بخش با واژه کلیدی **initialization** شروع شده و تا آغاز بخش اتمام (finalization) ادامه می‌یابد یا، چنان‌چه بخش اتمام (finalization) وجود نداشت، تا پایان یونیت ادامه پیدا می‌کند. بخش ارزش‌گذاری (initialization) شامل دستوراتی است که هنگام راه‌اندازی برنامه— به ترتیبی که دستورات ظاهر می‌شوند— اجرا می‌شوند. بنابراین، برای مثال، اگر شما ساختار داده‌هایی تعریف کرده باشید که لازم است مقداردهی اولیه شوند، می‌توانید این کار را در بخش ارزش‌گذاری (initialization) انجام دهید.

بخش‌های ارزش‌گذاری یونیت‌های به کار برده شده بوسیله یک مشتری، به همان ترتیبی که یونیت‌ها در شرط **uses** مشتری ظاهر می‌شوند، اجرا می‌شوند.

بخش اتمام (finalization)

بخش اتمام اختیاری است و تنها در یونیت‌هایی می‌تواند ظاهر شود که یک بخش ارزش‌گذاری (**initialization**) داشته باشند. بخش واسط با کلمه رزرو شده **finalization** آغاز شده و تا پایان یونیت ادامه می‌یابد. این بخش شامل دستوراتی است که هنگام خاتمه برنامه اصلی، اجرا می‌شوند. از این بخش برای آزادسازی منابع تخصیص داده شده در بخش ارزش‌گذاری استفاده کنید.

---

[۱] private

بخش‌های اتمام (**finalization**) عکس ترتیب ارزش‌گذاری‌ها (initializations) اجرا می‌شوند. برای مثال، اگر برنامه یونیت‌های A، B و C را به ترتیبی که آمد، مقدار دهی کند، بخش **finalization** آنها را به ترتیب C، B و A به پایان می‌رساند.

هروقت یک کد ارزش‌گذاری (**initialization**) شروع به اجرا کند، اجرای بخش اتمام (**finalization**) متناظر با آن در زمان بسته شدن برنامه، تضمین می‌شود. بخش اتمام (**finalization**) بایستی قادر به اداره کردن داده‌هایی که به طور ناتمام مقداردهی شده‌اند باشد، زیرا اگر یک خطای زمان اجرا رخ دهد، ممکن است که کد ارزش‌گذاری به طور کامل اجرا نشود.

ارجاعات یونیت و شرط uses

شرط **uses** یونیت‌هایی را که توسط برنامه، کتابخانه یا یونیت— جایی که در آنجا این شرط ظاهر شده— مورد استفاده قرار می‌گیرد، لیست می‌کند. (برای اطلاعات بیشتر درباره کتابخانه‌ها، «کتابخانه‌ها و بسته‌ها» را در فصل ۹ ملاحظه نمایید.)

شرط **uses** می‌تواند در موارد زیر ظاهر شود

- فایل پروژه مربوط به یک برنامه یا کتابخانه،
- بخش واسط (**interface**) یک یونیت، و
- بخش پیاده‌سازی (**implementation**) یک یونیت.

اغلب فایل‌های پروژه، همچون بخش‌های واسط اغلب یونیت‌ها، دارای یک شرط **uses** هستند. بعلاوه بخش پیاده‌سازی (**implementation**) یک یونیت می‌تواند شرط **uses** مال خود را داشته باشد. یونیت *System* به طور خودکار بوسیله هر پروژه ای مورد استفاده قرار می‌گیرد و نمی‌تواند به طور صریح در شرط **uses** لیست شود. (*System* روتین‌هایی را برای I/O فایل، اداره کردن رشته، عملیات‌های ممیز شناور، تخصیص حافظه پویا و ... پیاده‌سازی می‌کند.) یونیت‌های کتابخانه‌ای استاندارد دیگر، مانند *SysUtils* ، بایستی در شرط **uses** قرار گیرند. در اغلب موارد، زمانی که پروژه شما یک فایل منبع را تولید و پشتیبانی می‌کند، همه یونیت‌های لازم در شرط **uses** جای داده می‌شوند. برای اطلاعات بیشتر درباره جایابی و محتوای شرط **uses**، بخش‌های «ارجاعات غیرمستقیم و چندگانه یونیت» و «ارجاعات چرخشی یونیت» را در همین فصل ملاحظه نمایید.

قواعد شرط uses

شرط **uses** از کلمه رزرو شده **uses**، که پس از آن نام یک یا چند یونیت که با کاما از هم جدا شده و در آخر نیز یک نقطه ویرگول می‌آید، تشکیل می‌شود.

```
uses Forms, Main;
uses Windows, Messages, SysUtils, Strings, Classes, Unit2, MyUnit;
uses SysUtils, Types, Classes, QGraphics, QControls, QForms, QDialogs;
```

در شرط **uses** یک برنامه یا کتابخانه، ممکن است بعد از نام هر یونیت کلمه رزرو شده **in** و نام یک فایل منبع، با یا بدون یک مسیر دایرکتوری، در میان نشان نقل قول منفرد ' ' بیاید؛ مسیر دایرکتوری می‌تواند مطلق یا نسبی باشد. چند مثال:

```
uses Windows, Messages, SysUtils, Strings in 'C:\Classes\Strings.pas', Classes;
uses
QForms,
Main,
Extra in '../extra/extra.pas';
```

زمانی که نیاز به تعیین فایل منبع یونیت دارید، بعد از نام یونیت **in** قرار دهید. از آن جایی که IDE انتظار دارد که اسامی یونیت‌ها با اسامی فایل‌های منبع در جایی که آنها جای گرفته‌اند، مطابقت کنند، معمولاً دلیلی برای استفاده از **in** وجود ندارد. استفاده از **in** معمولاً زمانی لازم می‌شود که موقعیت فایل منبع مجهول و ناواضح باشد، برای مثال زمانی که

- از فایل منبعی استفاده کرده‌اید که در مسیر و دایرکتوری متفاوت از دایرکتوری فایل پروژه قرار دارد و این دایرکتوری در مسیر جستجوی کامپایلر یا کتابخانه عمومی قرار ندارد.
- دایرکتوری‌های متفاوتی در مسیر جستجوی کامپایلر، به طور یکسان یونیت‌ها را نام‌گذاری کرده‌اند.
- شما در حال کامپایل یک برنامه کنسول از خط فرمان هستید، و یک یونیت را با شناسه‌ای مشخص کرده‌اید که با نام فایل منبعش مطابقت ندارد.

همچنین کامپایلر به ساختمان ... **in** برای تعیین این که کدام یونیت‌ها بخشی از یک پروژه هستند، استناد می‌کند. تنها یونیت‌هایی که در شرط **uses** یک فایل پروژه (dpr.) همراه با **in** و یک نام فایل ظاهر می‌شوند، به عنوان بخشی از پروژه در نظر گرفته می‌شوند؛ یونیت‌های دیگر واقع در شرط **uses** بوسیله پروژه مورد استفاده قرار می‌گیرند، بدون این که به آن تعلق داشته باشند. این تمایز اثری بر روی کامپایل شدن ندارد، اما بر روی ابزار IDE مانند مدیر پروژه و مرورگر پروژه اثر می‌گذارد.

در شرط **uses** یک یونیت، نمی‌توانید از **in** استفاده کنید تا به کامپایلر بگویید که کجا یک فایل منبع را پیدا کند. هر یونیتی بایستی در مسیر جستجوی کامپایلر، مسیر جستجوی کتابخانه عمومی، یا همان مسیر که برای مثال یونیت از آن استفاده می‌کند، قرار داشته باشد. علاوه بر این، اسامی یونیت‌ها بایستی با اسامی فایل‌های منبع آنها مطابقت داشته باشند.

ارجاعات یونیت غیرمستقیم و چندگانه

ترتیبی که یونیت‌ها در شرط **uses** ظاهر می‌شوند، ترتیب ارزش‌گذاری (**initialization**) را تعیین می‌کند و بر شیوه‌ای که شناسه‌ها بوسیله کامپایلر مکان‌یابی می‌شوند، اثر می‌گذارد. درصورتی که دو یونیت یک متغیر، ثابت، نوع، روال یا تابع را با نام یکسانی اعلان کنند، کامپایلر از یکی که آخر از همه در شرط **uses** لیست شده، استفاده می‌کند. (برای دسترسی به شناسه از یونیت دیگر، شما می‌توانید یک توصیف کننده را به این طریق اضافه کنید: *UnitName.Identifier*.) شرط **uses** صرفاً لازم است یونیت‌هایی را در برداشته باشد که به طور مستقیم توسط برنامه یا یونیت— جایی که شرط ظاهر می‌شود— مورد استفاده قرار می‌گیرند. به عبارت دیگر، اگر یونیت A به ثابت‌ها، نوع‌ها، متغیرها، روال‌ها یا توابعی که در یونیت B اعلان شده اند، ارجاع بدهد، پس A بایستی به طور صریح از B استفاده کند. درصورتی که B به نوبت، به شناسه‌هایی از یونیت C ارجاع دهد، آنگاه A به طور غیر مستقیم به C وابسته است؛ در این مورد، C نیاز ندارد که در شرط **uses** مربوط به A جای گیرد، اما کامپایلر بایستی هنوز قادر باشد که به ترتیب B و C را برای پردازش A پیدا کند. مثال زیر وابستگی غیر مستقیم را نشان می‌دهد:

```
program Prog;
uses Unit2;
const a = b;
...
unit Unit2;
interface
uses Unit1;
const b = c;
...
unit Unit1;
interface
const c = 1;
...
```

دراین مثال، *Prog* مستقیماً به *Unit2* وابسته است در حالی که *Unit2* مستقیماً به *Unit1* وابسته است. از این رو *Prog* غیرمستقیم به *Unit1* وابسته است. از آن جایی که *Unit1* در شرط **uses** برای *Prog* ظاهر نمی‌شود، شناسه‌های اعلان شده در *Unit1* برای *Prog* غیرقابل دسترس هستند.

برای کامپایل یک واحد مشتری، کامپایلر نیازمند این است که همه یونیت‌هایی را که مشتری به آنها وابسته است، به طور مستقیم یا غیرمستقیم، مکان‌یابی کند. مگر این که کد منبع مربوط به این یونیت‌ها تغییر کرده باشد، اگر چه کامپایلر تنها فایل‌های dcu. (برای ویندوز) یا dcu/.dpu. (برای لینوکس) آنها را نیاز دارد، نه فایل‌های منبع (pas.) آنها را.

وقتی تغییرات در بخش واسط (**interface**) یک یونیت اعمال شود، یونیت‌های دیگری که به آن وابسته هستند، بایستی مجدداً کامپایل شوند. اما زمانی که تغییرات تنها در بخش پیاده‌سازی (**implementation**) یا بخش‌های دیگر یک یونیت اعمال می‌شوند، یونیت‌های وابسته لازم نیست که مجدداً کامپایل شوند. کامپایلر به طور خودکار این وابستگی‌ها را ردیابی کرده و یونیت‌ها را تنها زمانی کامپایل مجدد می‌کند که لازم باشد.

ارجاعات یونیت چرخشی

هنگامی‌که یونیت‌ها به طور مستقیم یا غیرمستقیم به یکدیگر ارجاع می‌دهند، گفته می‌شود که یونیت‌ها متقابلاً به یکدیگر وابسته هستند. وابستگی‌های متقابل تا زمانی مجاز هستند که مسیرهای چرخشی که شرط **uses** از یک بخش واسط (interface) را به شرط **uses** از یکی دیگر پیوند می‌دهد، وجود نداشته باشند. به عبارت دیگر، با شروع از بخش واسط یک یونیت، بایستی برگشت به آن یونیت با دنبال کردن ارجاعات از طریق بخش‌های واسط (interface) از یونیت‌های دیگر، امکان پذیر نباشد. برای یک الگو از وابستگی‌های متقابل که معتبر باشد، هر مسیر ارجاع چرخشی بایستی دست کم از طریق شرط **uses** یک بخش پیاده‌سازی (implementation) هدایت شود.

در ساده‌ترین حالت وابستگی متقابل دو یونیت، این حرف به معنای این است که یونیت‌ها نمی‌توانند همدیگر را در شرط‌های **uses** واسط‌های (interface) یکدیگر لیست کنند. بنابراین مثال زیر منجر به یک خطای کامپایل می‌شود:

```
unit Unit1;
interface
uses Unit2;
```

```
...
unit Unit2;
interface
uses Unit1;
...
```

گرچه، در صورتی که یکی از ارجاعات به بخش پیاده‌سازی (**implementation**) انتقال یابد دو یونیت می‌توانند قانوناً به یکدیگر ارجاع دهند:

```
unit Unit1;
interface
uses Unit2;
...
unit Unit2;
interface
...
implementation
uses Unit1;
...
```

برای کاهش احتمال ارجاعات چرخشی، هرزمان که امکان‌پذیر باشد، لیست کردن یونیت‌ها در شرط **uses** بخش پیاده‌سازی (**implementation**) ایده خوبی خواهد بود. تنها زمانی که شناسه‌هایی از یونیت دیگر در بخش واسط (**interface**) مورد استفاده قرار گیرند، لازم است که آن یونیت را در شرط **uses** بخش واسط (**interface**) لیست کرد.

فصل ۴

عناصر نحوی

پاسکال شیئی از مجموعه کاراکترهای اسکی (ASCII) استفاده می‌کند، کـه حـروف A تـا Z و a تـا z و ارقام 0 تا 9 ، و کاراکترهای استاندارد دیگر را دربردارد. پاسکال شیئی نسبت به حالت حروف حساس نیست. کاراکتر فضای خالی (ASCII 32) و کاراکترهای کنترلی (ASCII 0 تا 31 — ازجملـه ASCII 13، کاراکتر بازگشت یا انتهای خط) *کاراکترهای خالی*[۱] خوانده می‌شوند.

عناصر نحوی بنیادی، که *نشـانه‌ها*[۲] خوانده می‌شوند، با یکدیگر ترکیب می‌شوند تا عبارات، اعلان‌هـا و دستورات را شکل دهند. دستور[۳]، عملیات الگوریتمی را توصیف می‌کند که می‌تواند در میان یک برنامه اجرا شود. *عبارت*[۴] یک واحد نحوی است که در میان یک دستور ظاهر شـده و مقـداری را مشـخص می‌کند. یک *اعلان*[۵] شناسه‌ای (مانند اسم یک تابع یا متغیر) را تعریف می‌کند که می‌توانـد در عبـارات و دستورات مورد استفاده قرار گیرد و در جای مقتضی، حافظه لازم را برای شناسه تخصیص می‌دهد.

---

۱ blank

۲ tokens

۳ statement

۴ expression

۵ declaration

عناصر نحوی بنیادی

در ساده‌ترین سطح، یک برنامه دنباله‌ای از نشانه‌هاست که با تفکیک کننده‌ها از هم جدا می‌شوند. یک *نشانه* کوچکترین واحد قابل شمارش متن در یک برنامه است. یک تفکیک کننده یا یک جای خالی و یا یک توضیح است. اگر دقیق‌تر بگوییم، همیشه لازم نیست که یک تفکیک کننده بین دو نشانه قرار دهیم؛ برای مثال، قطعه کد زیر

```
Size:=20;Price:=10;
```

کاملاً مجاز است. اگرچه، قرارداد و خوانا بودن به ما دیکته می‌کند که آن را مانند زیر بنویسیم

```
Size := 20;
Price := 10;
```

نشانه‌ها به صورت علایم ویژه، شناسه‌ها، کلمات رزرو شده، راهنماها، اعداد، برچسب‌ها و رشته‌های کاراکتر طبقه بندی می‌شوند. یک تفکیک کننده تنها در صورتی می‌تواند بخشی از یک نشانه باشد که نشانه یک رشته کاراکتر باشد. شناسه‌ها، کلمات رزرو شده، اعداد و برچسب‌های مجاور بایستی باید یک یا چند تفکیک کننده مابین خود داشته باشند.

علایم ویژه

علایم ویژه[1] کاراکترهای غیرالفبایی، یا جفت‌هایی از این قبیل کاراکترها هستند، که معانی ثابتی دارند. تک تک کاراکترهای زیر علایم ویژه هستند.

```
# $ & ' ( ) * + , – . / : ; < = > @ [ ] ^ { }
```

جفت کاراکترهای زیر هم جزو علایم ویژه هستند.

```
(*  (.  *)  .)  ..  //  :=  <=  >=  <>
```

براکت چپ —**[**— با جفت کاراکتر پارانتزچپ و نقطه— **.(**— هم ارز است. براکت راست —**]**— با جفت کاراکتر نقطه و پارانتز راست— **.)**— هم ارز است. پارانتز چپ به اضافه ستاره و ستاره به اضافه پارانتز راست—**(*)**— با آکولادهای چپ و راست —**{ }**—هم ارز است.

[1] special symbols

توجه کنید که **!،"** (علامت نقل قول مضاعف)، **%**، **?**، \، _ (زیرخط)، **|** (میله)، و ~ (مد) جزو کاراکترهای خاص نیستند.

شناسه‌ها

شناسه‌ها[1] ثابت‌ها، متغیرها، فیلدها، نوع‌ها، خاصیت‌ها، توابع، برنامه‌ها، یونیت‌ها، کتابخانه‌ها و بسته‌ها را مشخص می‌کنند. یک شناسه می‌تواند هر طولی داشته باشد، اما تنها ۲۵۵ کاراکتر اول بامعنی هستند. یک شناسه باید با یک حرف یا یک زیرخط (_) شروع شود و نمی‌تواند شامل فضاهای خالی باشد؛ حروف، ارقام، و زیرخط‌ها بعد از اولین کاراکتر مجاز هستند. کلمات رزرو شده نمی‌توانند به عنوان شناسه‌ها مورد استفاده قرار گیرند.

از آن جایی که پاسکال شیئی نسبت به حالت حروف (بزرگ یا کوچک بودن) غیرحساس است، یک شناسه مانند *CalculateValue* می‌تواند به هر یک از شیوه‌های زیر نوشته شود:

```
CalculateValue
calculateValue
calculatevalue
CALCULATEVALUE
```

در محیط لینوکس، تنها برای شناسه‌هایی حالت (بزرگی یا کوچکی حروف) مهم است که اسامی یونیت باشند. از آن جایی که اسامی یونیت با اسامی فایل‌ها مطابقت می‌کنند، برخی اوقات ناسازگاری‌ها در حالت (بزرگی یا کوچکی حروف) می‌تواند بر کامپایل شدن اثر گذارد.

شناسه‌های توصیف شده

هرگاه از یک شناسه استفاده می‌کنید که در بیش از یک جا اعلان شده است، برخی اوقات لازم می‌شود که شناسه را قیددار کنید. ترکیب نوشتاری یک شناسه قیددار شده این گونه است

```
identifier1.identifier2
```

جایی که *identifier1* شناسه *identifier2* را توصیف می‌کند. برای مثال، چنان چه دو یونیت هر کدام یک متغیر با نام *CurrentValue* را اعلان کرده باشند، شما می‌توانید با نوشتن کد زیر تصریح کنید که می‌خواهید به *CurrentValue* در *Unit2* دسترسی پیدا کنید:

```
Unit2.CurrentValue
```

[1] Identifiers

قیدها می‌توانند تکرار شوند. برای مثال :

```
Form1.Button1.Click
```

متد *Click* را برای *Button1* متعلق به *Form1* فرامی‌خواند.

درصورتی که شما یک متغیر را مقید نکنید، تعبیر آن به وسیله قواعد دامنه که در بخش «بلوک‌ها و دامنه» شرح داده می‌شود، مشخص می‌شود.

کلمات رزرو شده

کلمات رزرو شده زیر، نمی‌توانند مجدداً تعریف شده یا به عنوان شناسه مورد استفاده قرار گیرند.

واژه‌های رزرو شده **Table 4.1**

and	**downto**	**in**	**or**	**string**
array	**else**	**inherited**	**out**	**then**
as	**end**	**initialization**	**packed**	**threadvar**
asm	**except**	**inline**	**procedure**	**to**
begin	**exports**	**interface**	**program**	**try**
case	**file**	**is**	**property**	**type**
class	**finalization**	**label**	**raise**	**unit**
const	**finally**	**library**	**record**	**until**
constructor	**for**	**mod**	**repeat**	**uses**
destructor	**function**	**nil**	**resourcestring**	**var**
dispinterface	**goto**	**not**	**set**	**while**
div	**if**	**object**	**shl**	**with**
do	**implementation**	**of**	**shr**	**xor**

علاوه بر کلمات جدول **4.1** ، **private**, **protected**, **public**, **published** و **automated** مابین اعلان‌های نوع شیء مانند کلمات رزرو شده عمل می‌کنند، در غیر این صورت با آنها مانند راهنماها برخورد می‌شود. کلمات **at** و **on** نیز معانی به خصوصی دارند.

راهنماها

راهنماها کلماتی هستند که در موقعیت‌های ویژه‌ای در میان کد منبع، دارای حساسیت هستند. راهنماها در پاسکال شیئی معانی خاصی دارند، اما برخلاف کلمات رزرو شده، تنها در بافت‌هایی ظاهر می‌شوند

که شناسه‌های تعریف شده توسط کاربر نمی‌توانند در آن جا ظاهر شوند. از این رو— اگر چه انجام این کار نارواست— شما می‌توانید شناسه‌ای تعریف کنید که دقیقاً شبیه یک راهنما باشد.

Table 4.2 **راهنماها**

absolute	**message**	**dynamic**	**private**	**resident**
abstract	**export**	**name**	**protected**	**safecall**
assembler	**external**	**near**	**public**	**stdcall**
automated	**far**	**nodefault**	**published**	**stored**
cdecl	**forward**	**overload**	**read**	**varargs**
contains	**implements**	**override**	**readonly**	**virtual**
default	**index**	**package**	**register**	**write**
deprecated	**library**	**pascal**	**reintroduce**	**writeonly**
dispid	**local**	**Platform**	**requires**	

ثابت‌های عددی (Numerals)

ثوابت صحیح و حقیقی می‌توانند در نمادگذاری اعشاری به صورت دنباله‌ای از ارقام بدون ویرگول‌ها یا فضاهای خالی، و نیز پیشوندار شده با عملگرهای + یا – به منظور نشان دادن علامت، نمایش داده شوند. مقادیر به طور پیش فرض مثبت هستند (طوری که، مثلاً، 67258 با 67258+ معادل است) و بایستی در میان دامنه بزرگترین نوع صحیح یا حقیقی از پیش تعیین شده باشد.

اعداد همراه با نقاط اعشاری یا نما اعداد حقیقی را مشخص می‌کنند، در حالی که اعداد دیگر صحیح‌ها را مشخص می‌کنند. زمانی که کاراکتر E یا e در میان یک عدد حقیقی ظاهر می‌شود، به معنی «ضرب در ده به قوه» می‌باشد. برای مثال 7E–2 به معنای 7×10^{-2} (یعنی هفت ضرب در ده به قوه منفی دو) است، و 12.25e+6 و 12.25e6 هردو به معنای 12.25×10^{6} هستند.

پیشوند علامت دلار ($) یک عدد مبنای شانزده را نشان می‌دهد— برای مثال، $8F. برای نوع صحیح (عدد صحیح ۱۶–بیتی)، علامت یک عدد مبنای شانزده بوسیله چپ‌ترین (با اهمیت‌ترین) بیت از نمایش باینری آن، تعیین می‌شود. برای همه انواع دیگر، شما بایستی از یک پیشوند + یا – برای بیان علامت، استفاده کنید.

برای اطلاعات بیشتر درباره انواع داده صحیح و حقیقی، بخش «انواع داده، متغیرها و ثابت‌ها»، را در فصل۵ ملاحظه نمایید. برای اطلاعات بیشتر درباره انواع داده ثوابت عددی، بخش «ثابت‌های صحیح» را در فصل۵ ملاحظه نمایید.

برچسب‌ها

برچسب[۱] دنباله‌ای از ارقام است که بیشتر از چهار رقم ندارد— یعنی یک عدد بین 0 تا 9999 است. صفرهای مقدم بامعنی نیستند. شناسه‌ها نیز می‌توانند به صورت برچسب عمل کنند. برچسب‌ها در دستورات **goto** به کار می‌روند. برای اطلاعات بیشتر درباره دستور **goto** و برچسب‌ها، «دستور **goto**» را در همین فصل ملاحظه نمایید.

رشته‌های کاراکتری

رشته کاراکتری، که یک *لیترال رشته* یا *ثابت رشته* نیز خوانده می‌شود، از یک رشته نقل قولی، یک رشته کنترل، یا ترکیبی از رشته‌های کنترل و نقل قولی، تشکیل می‌شود. تفکیک کننده‌ها تنها در میان رشته‌های نقل قولی رخ می‌دهند.

یک رشته نقل قولی دنباله‌ای از کاراکترها، حداکثر تا ۲۵۵ کاراکتر از مجموعه کاراکتر ASCII، نوشته شده در یک خط و محصور شده در میان آپوستروف‌هاست. یک رشته نقل قولی بدون هیچ چیز در میان آپوستروف‌ها یک رشته پوچ (*null*) می‌باشد. دو آپوستروف متوالی در یک رشته نقل قولی یک کاراکتر مجزا، یعنی یک آپوستروف، را معنی می‌دهد. برای مثال

```
'BORLAND'       { BORLAND }
'You''ll see'   { You'll see }
''''            { ' }
''              { null string }
' '             { a space }
```

رشته کنترلی دنباله ای از یک یا چند *کاراکتر کنترلی* است، که هر کدام از علامت # که پس از آن یک عدد صحیح بدون علامت از 0 تا 255 (دهدهی یا مبنای ۱۶) می‌آید، تشکیل می‌شود و کاراکتر ASCII متناظر را معنی می‌دهد. رشته کنترلی

```
#89#111#117
```

[۱] Label

با رشته نقل قولی زیر معادل است

```
'You'
```

شما می‌توانید رشته‌های نقل قولی را با رشته‌های کنترلی ترکیب کنید تا رشته‌های کاراکتری بزرگتری را شکل دهید. برای مثال، می‌توانید از کد زیر استفاده کنید تا

```
'Line 1'#13#10'Line 2'
```

یک سرسطر–تعویض‌سطر را میان "Line 1" و "Line 2" قرار دهید. اگر چه شما با این روش نمی‌توانید دو رشته نقل قولی را به هم دیگر الحاق کنید، از این رو یک جفت آپوستروف به صورت یک کاراکتر منفرد تفسیر می‌شود. (برای اتصال رشته‌های نقل قولی از عملگر + که در بخش «عملگرهای رشته‌ها» تشریح می‌شود، استفاده کنید، یا به سادگی آنها را در رشته نقل قولی واحدی ترکیب کنید.)

طول یک رشته کاراکتری برابر با تعداد کاراکترهای واقع در رشته است. یک رشته کاراکتری از هر طولی با هر نوع رشته و با نوع PChar سازگار است. یک رشـته کاراکتری با طول ۱ با هر نوع کاراکتر سازگار است، و زمانی که ترکیب نوشتاری بسط یافته فعال شده باشد (**{$X+}**)، یک رشته کاراکتر با طول $n \geq 1$ با آرایه‌های مبنای صفر و آرایه‌های بسته‌ای از n کاراکتر سازگار است. برای اطلاعات بیشتر درباره انواع رشته، بخش «انواع داده، متغیرها و ثابت‌ها» را در فصل۵ ملاحظه نمایید.

توضیحات و راهنماهای کامپایلر

کامپایلر از توضیحات صرفنظر می‌کند، به استثنای زمانی که آنها به صورت تفکیک کننده (تفکیک مرز نشانه‌های مجاور) یا راهنماهای کامپایلر عمل کنند. روش‌های متعددی برای ایجاد توضیحات وجود دارد:

```
{  Text between a left brace and a right brace constitutes a comment. }

(*  Text between a left-parenthesis-plus-asterisk and an
asterisk-plus-right-parenthesis also constitutes a comment. *)

 // Any text between a double-slash and the end of the line constitutes a comment
```

یک توضیح که حاوی یک علامت دلار ($) بلافاصله پس از آکولاد باز **{** یا **(*** باشد، یک راهنمای کامپایلر است. برای مثال،

```
{$WARNINGS  OFF}
```

به کامپایلر می‌گوید که پیغام‌های هشدار تولید نکند.

عبارت‌ها

یک عبارت (*expression*) ساختاری است که مقداری را برمی‌گرداند. برای مثال:

```
X                { variable }
@X               { address of a variable }
15               { integer constant }
InterestRate     { variable }
Calc(X,Y)        { function call }
X * Y            { product of X and Y }
Z / (1 - Z)      { quotient of Z and (1 - Z) }
X = 1.5          { Boolean }
C in Range1      { Boolean }
not Done         { negation of a Boolean }
[ 'a' , 'b' , 'c' ]  { set }
Char(48)         { value typecast }
```

ساده‌ترین عبارت‌ها، متغیرها و ثابت‌ها هستند (تشریح شده در فصل۵، بخش «انواع داده، متغیرها و ثابت‌ها»). عبارات پیچیده‌تر با استفاده از عملگرها، فراخوان‌های توابع، سازنده‌های مجموعه، اندیس‌ها و قالب‌بندی‌های نوع[۱]، از عبارات ساده‌تر ساخته می‌شوند.

عملگرها

عملگرها که جزیی از زبان پاسکال شیئی هستند، مانند توابع از پیش تعریف شده‌ای عمل می‌کنند. برای مثال، عبارت (X + Y) از متغیرهای X و Y — که عملوند خوانده می‌شوند— و عملگر + ساخته می‌شود؛ زمانی که X و Y بیانگر مقادیر صحیح یا حقیقی باشند، (X + Y) مجموع آنها را برمی‌گرداند. عملگرها شامل **@, not, ^, *, /, div, mod, and, shl, shr, as, +, –, or, xor, =, >, <, <>, <=, >=, in** و **is** هستند.

عملگرهای **@**، **not** و **^** *یکانی* هستند (یعنی یک عملوند می‌گیرند). تمامی‌عملگرهای دیگر *باینری* هستند (یعنی دو عملوند می‌گیرند)، به استثنای **+** و **–** که یا به صورت یکانی عمل می‌کنند یا باینری. یک عملگر یکانی معمولاً مقدم بر عملوند خود است (برای مثال، B-) به استثنای ^ که بعد از عملوند

---

[۱] Type Casting

خود می‌آید (برای مثال، P^). یک عملگر باینری مابین دو عملوند خود قرار می‌گیرد (برای مثال، A=7).

برخی عملگرها بسته به نوع داده‌ای که با آن سروکار دارند، رفتار متفاوتی از خود نشان می‌دهند. برای مثال، **not** به صورت نفی بیتی بر روی یک عملوند صحیح و نفی منطقی برروی یک عملوند بولی عمل می‌کند. چنین عملگرهایی در زیر، تحت چندین دسته ظاهر می‌شوند. به استثنای ^ ، **is** و **in** ، همه عملگرها می‌توانند عملوندهایی از نوع *واریانت*[1] بگیرند. برای دیدن جزییات، بخش «انواع واریانت» را در فصل۵ ملاحظه نمایید.

در بخش‌هایی که متعاقباً می‌آید، فرض بر این است که برخی آشنایی‌ها با انواع داده در پاسکال شیئی وجود دارد. برای اطلاعات بیشتر درباره انواع داده، فصل۵ را ملاحظه نمایید. برای اطلاعات بیشتر درباره حق تقدم عملگرها در عبارات پیچیده، بخش «قواعد حق تقدم عملگر» را در همین فصل ملاحظه کنید.

عملگرهای محاسباتی

عملگرهای محاسباتی که عملوندهای صحیح یا حقیقی می‌گیرند، شامل +، **–** ، ***، /، div** و **mod** هستند.

Table 4.3 عملگرهای محاسباتی باینری

عملگر	عملیات	نوع عملوند	نوع نتیجه	مثال
+	جمع	integer, real	integer, real	X + Y
–	تفریق	integer, real	integer, real	Result - 1
*	ضرب	integer, real	integer, real	P * InterestRate
/	تقسیم حقیقی	integer, real	real	X / 2
div	تقسیم صحیح	integer	integer	Total **div** UnitSize
mod	باقیمانده	integer	integer	Y mod 6

[1] Variant

عملگرهای محاسباتی یکانی **Table 4.4**

عملگر	عملیات	نوع عملوند	نوع نتیجه	مثال
+	شناسایی علامت	integer, real	integer, real	+7
–	نفی علامت	integer, real	integer, real	-X

قواعد زیر به عملگرهای محاسباتی اعمال می‌شوند.

- مقدار x/y علیرغم نوع x و y ، از نوع *Extended* است. برای عملگرهای محاسباتی دیگر، زمانی که حداقل یکی از عملوندها از نوع حقیقی باشد، نتیجه از نوع *Extended* خواهد بود؛ درغیر این صورت، زمانی که حداقل یکی از عملوندها از نوع *Int64* باشد، نتیجه از نوع *Int64* خواهد شد؛ در غیر این صورت نتیجه از نوع *Integer* است. در صورتی که نوع یک عملوند یک زیردامنه نوع صحیح باشد، با آن طوری رفتار می‌شود که انگار نوع آن صحیح بوده است.
- مقدار *x* **div** *y* برابر مقدار *x*/*y* گرد شده در امتداد صفر تا نزدیک ترین عدد صحیح است.
- عملگر **mod** باقیمانده حاصل از تقسیم عملوندهایش را برمی‌گرداند. به عبارت دیگر . *x* **mod** *y* = *x* – (*x* **div** *y*) * *y*
- در عباراتی به فرم *x*/*y* ، *x* **div** *y* یا *x* **mod** *y* زمانی که *y* برابر صفر باشد، یک خطای زمان اجرا رخ می‌دهد.

عملگرهای بولی

عملگرهای بولی[1] **not**، **and**، **or** و **xor** عملوندهایی از هر نوع بولی را گرفته و یک مقدار از نوع *Boolean* را برمی‌گردانند.

عملگرهای بولی **Table 4.5**

عملگر	عملیات	نوع عملوند	نوع نتیجه	مثال

[1] Boolean Operator

not	نفی	*Boolean*	*Boolean*	**not** (C **in** MySet)
and	ترکیب عطفی	*Boolean*	*Boolean*	**and** (Total > 0)
or	فصل	*Boolean*	*Boolean*	A **or** B
xor	یای مانع جمع	*Boolean*	*Boolean*	A **xor** B

این عملیات‌ها بوسیله قواعد استاندارد منطق بولی کنترل می‌شوند. برای مثال، عبارتی بـه فـرم *x* **and** *y* برابر True است اگر و تنها اگر هر دوی *x* و *y* برابر True باشند.

ارزشیابی بولی کامل در برابر ارزشیابی مدار کوتاه

کامپایلر دو شیوه ارزشیابی را بـرای عملگرهـای **and** و **or** پشـتیبانی مـی‌کنـد: ارزشـیابی کامـل و ارزشیابی مدار کوتاه (جزیی). ارزشیابی کامل به این معنی است که هر متحد یـا غیـر متحـدی ارزیـابی می‌شود، حتی زمانی که نتیجه سرتاسر عبارت از قبل مشخص شده باشد. ارزیابی مدار کوتاه (جزیـی)، ارزیابی صریح چپ به راست را معنی می‌دهد که به محـض مشـخص شـدن نتیجـه سرتاسـر عبـارت، ارزشیابی متوقف می‌شود. برای مثال، در صورتی کـه عبـارت A **and** B تحـت ارزشـیابی مـدار کوتـاه ارزیابی شود، زمانی که A نادرست (False) باشد، کامپایلر دیگـر B را ارزیـابـی نخواهـد کـرد، و بـه محض اینکه A را ارزیابی می‌کند، دیگر می‌داند که سرتاسر عبارت نادرست (False) است.

ارزشیابی مدار کوتاه ارجح‌تر است زیرا حداقل زمان اجرا را متقبل می‌شود و در اغلب موارد، حداقل اندازه کد را می‌گیرد. گاهی، زمانی که یکی از عملوندها یک تابع با اثرات جنبی باشد که اجرای برنامه را دگرگون می‌کند، ارزشیابی کامل مناسب‌تر است. همچنین ارزشیابی مدار کوتاه اجازه استفاده از ترکیب‌ها و ساختارهایی را می‌دهد که می‌توانند نتایج دیگری در عملیات‌های زمان اجرای غیر مجاز داشته باشند. برای مثال، کد زیر سرتاسر رشته S را جستجو می‌کند تا به اولین ویرگول برسد.

```
while (I <= Length(S)) and (S[I] <> ',') do
begin
...
Inc(I);
end;
```

در حالتی که *S* ویرگولی نداشته باشد، تکرار آخری I را به مقداری که بزرگتر از طول S اسـت افـزایش می‌دهد. زمانی که شرط **while** بعداً تست شود، نتیجه ارزشیابی کامل یک تلاش برای خواندن *S[I]* را نتیجه می‌دهد، که منجر به بروز یک خطای حین اجرا خواهد شد. در مقابل، تحت ارزشیابی مدار کوتـاه

دومین بخش از شرط **while** —(',' <> S[I])— بعد از اینکه بخش اول نافرجام بماند، ارزیابی نمی‌شود.

از راهنمای کامپایلر **$B** برای کنترل شیوه ارزشیابی استفاده کنید. حالت پیش فرض **{$B–}** است، که ارزشیابی اتصال کوتاه را فعال می‌کند. برای فعال کردن ارزشیابی کامل به صورت موضعی، فرمان **{$B+}** را به کد خود اضافه کنید. همچنین شما می‌توانید با انتخاب ارزشیابی بولی کامل در جعبه گفتگوی گزینه‌های کامپایلر (Compiler Options)، به ارزشیابی کامل در روی گستره یک پروژه سوییچ کنید.

توجه در صورتی که هر یک از عملوندهای درگیر یک واریانت باشد، کامپایلر همواره ارزشیابی کامل را انجام می‌دهد (حتی در حالت **{$B–}**).

عملگرهای منطقی بیتی

عملگرهای منطقی زیر، دستکاری بیتی را روی عملوندهای صحیح (integer) انجام می‌دهند. برای مثال، اگر مقدار ذخیره شده در X (در حالت باینری) برابر 001101 باشد و مقدار ذخیره شده در Y برابر 100001 باشد، عبارت

```
Z := X or Y;
```

مقدار 101101 را به Z تخصیص می‌دهد.

عملگرهای منطقی (بیتی) **Table 4.6**

عملگر	عملیات	نوع عملوند	نوع نتیجه	مثال
not	نفی بیتی	integer	integer	**not** X
and	بیتی **and**	integer	integer	X **and** Y
or	بیتی **or**	integer	integer	X **or** Y
xor	بیتی **xor**	integer	integer	X **xor** Y
shl	شیفت چپ بیتی	integer	integer	X **shl** 2
shr	شیفت راست بیتی	integer	integer	Y **shr** I

قواعد زیر به عملگرهای بیتی اعمال می‌شوند.

- نتیجه یک عملیات **not** از همان نوعی است که عملوند می‌باشد.
- اگر هردو عملوند یک عملیات **and**، **or** یا **xor**، از نوع صحیح باشند، نتیجه از نوع صحیح از پیش تعریف شده با کوچک ترین دامنه‌ای است که همه مقادیر امکان پذیر هر دو نوع را شامل باشد.
- عملیات‌های *x* **shl** *y* و *x* **shr** *y* مقدار *x* را به اندازه *y* بیت به سمت راست یا چپ جابه‌جا می‌کند (شیفت می‌کند) که هم ارز با این است که *x* را به *2y* ضرب یا تقسیم کنیم؛ نتیجه از همان نوع *x* خواهد بود. برای مثال، در صورتی که N مقدار 01101 (معادل مقدار دهدهی ۱۳) را ذخیره کند، آنگاه N **shl** 1 مقدار 11010 (معادل ۲۶ دهدهی) را برمی‌گرداند. توجه کنید که مقدار *y* به پیمانه اندازه نوع *x* تفسیر می‌شود. بنابراین برای مثال، چنان چه x یک عدد صحیح (integer) باشد، x **shl** 40 به صورت x **shl** 8 تفسیر می‌شود زیرا یک عدد صحیح integer ۳۲ بیتی است و 32 **mod** 40 برابر ۸ است.

عملگرهای رشته

عملگرهای رابطه‌ای =، <>، <، >، =< و >= همگی عملوندهای رشته‌ای را می‌گیرند. (بخش «عملگرهای رابطه‌ای» را در همین فصل ملاحظه نمایید.) عملگر + دو رشته را به هم متصل می‌کند.

Table 4.7 عملگرهای رشته

عملگر	عملیات	نوع عملوند	نوع نتیجه	مثال
+	اتصال دو رشته	string, packed string, character	string	S + '. '

قواعد زیر به الحاق رشته‌ها اعمال می‌شوند.

- عملوندها برای + می‌توانند رشته‌ها (strings)، رشته‌های بسته بندی شده (آرایه‌های بسته ای از نوع *Char*) یا کاراکترها باشند. اگرچه، در صورتی که یکی از عملوندها از نوع *WideChar* باشد، عملوند دیگر بایستی یک رشته بلند باشد.

نتیجه عملیات + (الحاق رشته‌ها) با هر نوع رشته‌ای سازگار است. گرچه، چنان چه هردو عملوند، رشته کوتاه یا کاراکتر باشند، و طول ترکیبی آنها بزرگتر از ۲۵۵ باشد، نتیجه به ۲۵۵ کاراکتر اول بریده می‌شود.

عملگرهای اشاره‌گر

عملگرهای رابطه‌ای <، >، =< و => می‌توانند عملوندهایی از نوع *PChar* بگیرند (بخش «عملگرهای رابطه‌ای» را در همین فصل ببینید). عملگرهای زیر نیز اشاره‌گرها را به عنوان عملوند می‌گیرند. برای اطلاعات بیشتر درباره اشاره‌گرها، بخش «اشاره‌گرها و انواع اشاره‌گر» را در فصل۵ ملاحظه نمایید.

Table 4.8 عملگرهای اشاره‌گر کاراکتری

عملگر	عملیات	نوع عملوند	نوع نتیجه	مثال
+	جمع اشاره‌گر	character pointer, integer	character pointer	P + I
–	تفریق اشاره‌گر	character pointer, integer	character pointer, integer	P - Q
^	برگشت ارجاع اشاره‌گر	pointer	base type of pointer	P^
=	تساوی	pointer	*Boolean*	P = Q
<>	نابرابری	pointer	*Boolean*	P <> Q

عملگر ^ یک اشاره‌گر را برگشت ارجاع می‌دهد. عملوند ^ می‌تواند از هر نوعی به استثنای اشاره‌گر عمومی *Pointer* باشد، که این نوع بایستی قبل از برگشت ارجاع دادن، قالب‌بندی[۱] شود.

تنها در حالتی *P = Q* درست (*True*) است که *P* و *Q* به آدرس یکسانی اشاره کنند، در غیر این صورت *P <> Q* درست است. شما می‌توانید از + و – برای افزایش یا کاهش آفست یک اشاره‌گر

[۱] تبدیل نوع صریح (Cast)

کاراکتر استفاده کنید. همچنین می‌توانید از – برای محاسبه اختلاف آفست‌های دو اشاره‌گر کاراکتر استفاده کنید. قواعد زیر قابل اعمال هستند.

- اگر *I* یک عدد صحیح و *P* یک اشاره‌گر کاراکتر باشد، آنگاه *P + I*، *I* را به آدرس داده شده بوسیله *P* اضافه می‌کند؛ یعنی این که یک اشاره‌گر به آدرس *I* کاراکتر بعد از *P* را برمی‌گرداند. (عبارت I + P با *P + I* معادل است.) عبارت *P - I*، *I* را از آدرس داده شده بوسیله *P* کم می‌کند؛ یعنی این که یک اشاره‌گر به *I* کاراکتر قبل از *P* را برمی‌گرداند.
- در صورتی که *P* و *Q* هردو اشاره‌گرهای کاراکتر باشند، آنگاه *P – Q* اختلاف بین آدرس داده شده بوسیله *P* (آدرس بالاتر) و آدرس داده شده توسط *Q* (آدرس پایین‌تر) را محاسبه می‌کند؛ یعنی این که یک عدد صحیح دال بر تعداد کاراکترها میان *P* و *Q* را برمی‌گرداند. *P + Q* تعریف نشده است.

عملگرهای مجموعه

عملگرهای زیر مجموعه‌ها را به عنوان عملوند می‌پذیرند.

عملگرهای مجموعه **Table 4.9**

عملگر	عملیات	نوع عملوند	نوع نتیجه	مثال
+	اجتماع	مجموعه	مجموعه	Set1 + Set2
–	تفاضل	مجموعه	مجموعه	S - T
*	اشتراک	مجموعه	مجموعه	S * T
<=	زیرمجموعه	مجموعه	*Boolean*	Q <= MySet
>=	ابرمجموعه	مجموعه	*Boolean*	S1 >= S2
=	تساوی	مجموعه	*Boolean*	S2 = MySet
<>	نابرابری	مجموعه	*Boolean*	MySet <> S1
in	عضویت	مجموعه، ترتیبی	*Boolean*	A **in** Set1

قواعد زیر به + ، – و * در نقش عملگرهای مجموعه اعمال می‌شوند.

- عدد ترتیبی O در X + Y است اگر و تنها اگر O در X یا Y باشد (یـا هـر دو). O در X – Y است اگر و تنها اگر O در X باشد اما در Y نباشد. O در X*Y است اگر و تنها اگر O هـم در X و هم در Y باشد.
- نتیجه یک عملیات +، – یا * از نوع **set of** *A..B* است، جایی کـه *A* کوچـک‌تـرین مقـدار ترتیبی در مجموعه حاصل و *B* بزرگترین‌اش است.

قواعد زیر به =>، =<، =، <> و **in** در نقش عملگرهای مجموعه، اعمال می‌شوند.

- X => Y درست است تنها در حالتی که تمامی اعضای X عضوی از Y باشند؛ Z >= W معادل با W <= Z است
- U = V تنها در حالتی درست است که U و V دقیقاً اعضای یکسانی داشته باشند؛ در غیر این صورت، U <> V درست است.
- برای عدد ترتیـبی O و یک مجموعه S، O **in** S تنها در حالتی *True*است که O عضوی از S باشد.

عملگرهای رابطه ای

عملگرهای رابطه‌ای برای مقایسه دو عملوند مورد استفاده قرار می‌گیرند. در ضمن عملگرهای =، <>، => و =< به مجموعه‌ها نیز اعمال می‌شوند (بخش «عملگرهای مجموعه» را ملاحظه نمایید)؛ = و <> به اشاره‌گرها نیز اعمال می‌شوند(بخش «عملگرهای اشاره‌گر» را ملاحظه نمایید).

Table 4.10 عملگرهای رابطه‌ای

عملگر	عملیات	نوع عملوند	نوع نتیجه	مثال
=	تساوی	simple, class, class reference, interface, string, packed string	*Boolean*	I = Max
<>	نابرابری	simple, class, class reference, interface, string, packed string	*Boolean*	X <> Y
<	کمتر از	simple, string, packed string, *PChar*	*Boolean*	X < Y
>	بیشتر از	simple, string, packed string, *PChar*	*Boolean*	Len > 0

Cnt <= I	*Boolean*	simple, string, packed string, *PChar*	کمتر از یا مساوی	<=
I >= 1	*Boolean*	simple, string, packed string, *PChar*	بیشتر از یا مساوی	>=

برای اغلب نوع‌های ساده، مقایسه سرراست و آسان می‌باشد. برای مثال، I = J تنها در حالتی درست است که I و J مقادیر یکسان داشته باشند، و در غیر این صورت I <> J درست است. قواعد زیر به عملگرهای رابطه‌ای اعمال می‌شوند.

- عملوندها بایستی از انواع سازگار باهم باشند، به استثنای این که یک عدد حقیقی و یک عدد صحیح می‌توانند با هم مقایسه شوند.
- رشته‌ها مطابق با ترتیب مجموعه کاراکتر ASCII بسط یافته مقایسه می‌شوند. انواع کاراکتر مانند رشته‌هایی با طول یک رفتار می‌کنند.
- دو رشته بسته‌بندی شده، بایستی تعداد عناصر یکسانی داشته باشند تا مقایسه شوند. زمانی که یک رشته بسته‌بندی شده با n عنصر با یک رشته مقایسه شود، رشته بسته‌بندی شده همانند یک رشته با طول n رفتار می‌کند.
- عملگرهای <، >، =< و => تنها در صورتی به عملوندهای PChar اعمال می‌شوند که دو اشاره‌گر به میان آرایه کاراکتر یکسانی اشاره کنند.
- عملگرهای = و <> می‌توانند عملوندهایی از نوع کلاس و class-reference بگیرند. با عملوندهایی از نوع کلاس، عملگرهای = و <> مطابق قواعدی که به اشاره‌گرها اعمال می‌شوند، ارزیابی می‌شوند: C = D تنها در حالتی درست است که C و D به وهله شیء یکسانی اشاره کنند، و در غیر این صورت C <> D درست است. با عملوندهایی از نوع class-reference، C = D تنها در حالتی درست است که C و D کلاس یکسانی را مشخص کنند، و در غیر این صورت C <> D درست است. برای اطلاعات بیشتر درباره کلاس‌ها فصل۷ را ملاحظه نمایید.

عملگرهای کلاس

عملگرهای **as** و **is** کلاس‌ها و وهله‌های اشیاء را به عنوان عملوند می‌پذیرند ؛ **as** روی واسط[۱]ها به خوبی عمل می‌کند. برای اطلاعات بیشتر «کلاس‌ها و اشیاء» را در فصل۷ و «واسط‌های شیء» را در فصل۱۰ ملاحظه نمایید.

درضمن عملگرهای رابطه‌ای = و <> هم روی کلاس‌ها عمل می‌کنند. بخش «عملگرهای رابطه‌ای» را در همین فصل ملاحظه نمایید.

عملگر @

عملگر@ آدرس یک متغیر، تابع، روال یا متد را برمی‌گرداند ؛ به عبارت دیگر، @ یک اشاره‌گر به عملوند خود ایجاد می‌کند. برای اطلاعات بیشتر درباره اشاره‌گرها، «اشاره‌گرها و انواع اشاره‌گر» را در فصل۵ ملاحظه نمایید. قواعد زیر به @ اعمال می‌شوند.

- اگر X یک متغیر باشد، X@ آدرس X را برمی‌گرداند. (چنان چه X یک متغیر رویه‌ای باشد، قواعد ویژه‌ای اعمال می‌شود؛ بخش «انواع رویه‌ای در دستورات و عبارات» را در فصل ۵ ملاحظه نمایید.) در صورتی که راهنمای کامپایلر**{$T–}** در حال اثر باشد، نوع X@ از نوع Pointer خواهد شد. در حالت **{$T+}**، X@ از نوع ^T خواهد بود، جایی که T از نوع X است.
- اگر F یک روتین باشد (یک تابع یا روال)، F@ نقطه ورود[۲] F را برمی‌گرداند. نوع F@ همواره *Pointer* خواهد بود.
- زمانی که @ به یک متد تعریف شده در یک کلاس اعمال شود، شناسه متد بایستی با نام کلاس قیددار شود. برای مثال،

---

[۱] Interface

[۲] Entry point

```
@TMyClass.DoSomething
```

به متد DoSomething از TMyClass اشاره می‌کند. برای اطلاعات بیشتر درباره کلاس‌ها و متدها فصل ۷ را ملاحظه نمایید.

قواعد حق تقدم عملگرها

در عبارات پیچیده، قواعد حق تقدم[۱] ترتیب انجام عملیات‌ها را تعیین می‌کنند. در جدولی که متعاقباً می‌آید، حق تقدم عملگرها جمع‌بندی شده است.

Table 4.11 حق تقدم عملگرها

حق تقدم	عملگر
اولین (بالاترین)	**@, not**
دومین	***, /, div, mod, and, shl, shr, as**
سومین	**+, –, or, xor**
چهارمین (پایین ترین)	**=, <>, <, >, <=, >=, in, is**

عملگری با حق تقدم بالاتر قبل از یک عملگر با حق تقدم پایین‌تر ارزیابی می‌شود، در حالی که عملگرهای با حق تقدم مساوی الحاق به چپ می‌شوند. از این رو عبارت

```
X + Y * Z
```

Y را به Z ضرب کرده، آنگاه X را به نتیجه اضافه می‌کند؛ در ابتدا * انجام می‌شود، زیرا حق تقدم بالاتری از + دارد. اما

```
X - Y + Z
```

نخست Y را از X کم می‌کند، آنگاه نتیجه را با Z جمع می‌کند ؛ – و + حق تقدم یکسانی دارند، بنابراین عملیات چپی زودتر انجام می‌گیرد (الحاق به چپ).

---

[۱] Precedence

برای لغو این قواعد، شما می‌توانید از پارانتزها استفاده کنید. در ابتدا عبارت داخل پارانتز ارزیابی می‌شود و سپس نتیجه همانند یک عملوند واحد عمل می‌کند. برای مثال

```
(X + Y) * Z
```

ابتدا X و Y باهم جمع شده و نتیجه در Z ضرب می‌شود.

پارانتزها گاهی اوقات در موقعیت‌هایی، که در نظر اول به نظر می‌رسد که نیازی به آنها نیست، لازم می‌شوند. برای مثال، به این عبارت توجه کنید

```
X = Y  or  X = Z
```

تعبیر مورد نظر این عبارت به طور واضح این است

```
( X = Y ) or ( X = Z )
```

اما بدون پارانتزها، کامپایلر از قواعد حق تقدم عملگرها پیروی کرده و آن را همچون عبارت زیر می‌خواند

```
( X = ( Y or X ) ) = Z
```

—که منجر به یک خطای کامپایل می‌شود مگر اینکه Z بولی (*Boolean*) باشد.

پارانتزها اغلب اوقات کدها را برای نوشتن و خواندن راحت‌تر می‌کنند، بخواهیم دقیق‌تر بگوییم، حتی اگر زمانی که آنها ظاهراً زیادی و غیرضروری باشند. بنابراین اولین مثال بالا می‌تواند مانند زیر نوشته شود

```
X + (Y * Z)
```

در اینجا پارانتزها برای کامپایلر غیر ضروری هستند، اما برنامه‌نویس و خواننده کد را از دانستن حق تقدم عملگرها بی‌نیاز می‌کنند.

فراخوان‌های تابع

از آن جایی که توابع مقداری را برمی‌گردانند، فراخوانی‌های توابع جزء عبارات محسوب می‌شوند. برای مثال، درصورتی که یک تابع تعریف کرده باشید که دو عدد صحیح را به عنوان آرگومان گرفته و یک عدد صحیح را برمی‌گرداند، آنگاه فراخوانی تابع Calc(24, 47) یک عبارت صحیح است. در صورتی که

I و J متغیرهای صحیح باشند، آنگاه I + Calc(J, 8) نیز یک عبارت صحیح خواهد بود. نمونه‌هایی برای فراخوانی توابع

```
Sum( A , 63 )
Maximum( 147 ,  J )
Sin( X + Y )
Eof( F )
Volume( Radius , Height )
GetValue
TSomeObject.SomeMethod( I , J );
```

برای اطلاعات بیشتر درباره توابع فصل۶، «روال‌ها و توابع» را ملاحظه نمایید.

سازنده‌های مجموعه

سازنده مجموعه مقدار یک نوع مجموعه (set-type) را تعیین می‌کند. برای مثال،

```
[5, 6, 7, 8]
```

مجموعه‌ای را که اعضایش 5، 6 ، 7 و 8 باشد، مشخص می‌کند. در ضمن سازنده مجموعه

```
[ 5..8 ]
```

همان مجموعه قبلی را مشخص می‌کند. ترکیب نوشتاری سازنده یک مجموعه به صورت زیر است:

```
[ item1, ..., itemn ]
```

جایی که *item* یا عبارتی است که یک عنصر ترتیبی از نوع اصلی مجموعه را مشخص می‌کند یا یک جفت از همچو عباراتی با دو نقطه (**..**) در بین‌شان. زمانی که یک *item* فرم **x..y** را داشته باشد، این فرم یک شکل خلاصه‌نویسی فراگیر برای همه عناصر ترتیبی در دامنه x تا y می‌باشد؛ اما در صورتی که x بزرگتر از y باشد، آنگاه **x..y** هیچ چیزی را مشخص نمی‌کند و **[*x..y*]** یک مجموعه تهی خواهد بود. سازنده مجموعه **[]** مجموعه تهی را مشخص می‌کند در حالی که **[*x*]** مجموعه‌ای را که تنها یک عضو دارد و مقدار آن هم x است، مشخص می‌کند. نمونه‌هایی برای سازنده‌های مجموعه‌ها:

```
[ red , green , MyColor ]
[1 , 5 , 10..K mod 12 , 23 ]
[ 'A'..'Z' , 'a'..'z' , Chr(Digit + 48) ]
```

برای اطلاعات بیشتر درباره مجموعه‌ها، بخش «مجموعه‌ها» را در فصل۵ ملاحظه نمایید.

اندیس‌ها

رشته‌ها، آرایه‌ها، خاصیت‌های آرایه و اشاره‌گرهایی بـه رشته‌ها یا آرایه‌ها می‌توانند انـدیس‌دار شـوند. برای مثال، چنان چه *FileName* یک متغیر رشته باشد، عبارت FileName[3] سـومین کـاراکتر واقـع در رشته تعیین شده بوسیله *FileName* را برمی‌گرداند، در حالی که FileName[I + 1] کاراکتری را بلافاصله یکی بالاتر از I اُمین کاراکتر برمی‌گرداند. برای اطلاعات بیشتر درباره رشته‌ها، بخش «انواع رشـته‌هـا» را در فصل۵ ملاحظه نمایید. برای اطلاعـات بیشـتر دربـاره آرایـه‌هـا و خاصـیت‌هـای آرایـه، بخـش «آرایه‌ها» را در فصل۵ و بخش «خواص آرایه‌ها» را در فصل۷ ملاحظه نمایید.

قالب‌بندی نوع (تبدیل نوع صریح)

گاهی سودمند است که با یک عبارت طوری رفتار کنیم، مثل این که به نوع متفاوتی تعلق دارد. قالب‌بندی نوع (یا تبدیل نوع صریح) در واقع به شما اجازه می‌دهد چنین کاری را، به طور موقت با تغییر نوع یک عبارت، انجام دهید. مثلاً، Integer('A') کاراکتر A را به عنوان یک عدد صحیح integer قالب‌بندی می‌کند.

ترکیب نوشتاری/نحوی قالب‌بندی نوع (تبدیل نوع صریح) به شکل زیر است

```
typeIdentifier(expression)
```

در صورتی که عبارت (*expression*) یک متغیر باشد، نتیجه یک قالب‌بندی نوع متغیر خوانده مـی‌شـود؛ در غیر این صورت، نتیجه یک قالب‌بندی مقدار است. در حالی که ترکیب نوشتاری آنها یکی است، امـا قواعد متفاوتی به دو نوع قالب‌بندی نوع اعمال می‌شود.

قالب‌بندی نوع مقدار

در قالب‌بندی نوع مقداری، شناسه نوع و عبارت قالب‌بندی بایستی هردو یا انواع ترتیبی باشند یا هردو انواع اشاره‌گر. مثال‌هایی برای قالب‌بندی نوع مقدار:

```
Integer('A')
Char(48)
Boolean(0)
Color(2)
Longint(@Buffer)
```

مقدار حاصل بوسیله تبدیل عبارت داخل پارانتزها، بدست می‌آید. چنان چه اندازه نوع تعیین شده با اندازه نوع عبارت فرق داشته باشد، این کار ممکن است درگیر عمل برش یا بسط گردد. علامت عبارت همواره حفظ می‌شود. دستور

```
I := Integer('A');
```

مقدار Integer('A') را ـــ که 65 است ـــ به متغیر I تخصیص می‌دهد.

قالب‌بندی نوع مقداری نمی‌تواند با توصیف کننده‌ها پی گرفته شود و ضمناً نمی‌تواند در سمت چپ یک دستور تخصیصی ظاهر شود.

قالب‌بندی نوع متغیر

شما می‌توانید هر متغیری را به هر نوعی قالب‌بندی کنید، اندازه‌های عرضه شده آنها یکسان است و اعداد صحیح را با اعداد حقیقی ترکیب نکنید. (برای تبدیل انواع عددی، به توابع استاندارد مانند *Int* و *Trunc* تکیه کنید.) نمونه‌ها :

```
Char(I)
Boolean(Count)
TSomeDefinedType(MyVariable)
```

قالب‌بندی نوع متغیر می‌تواند در هر کدام از طرفین یک دستور تخصیص ظاهر شود. بنابراین

```
var MyChar: char;
...
Shortint(MyChar) := 122;
```

کاراکتر(ASCII 122) z را به *MyChar* تخصیص می‌دهد.

شما می‌توانید متغیرها را به یک نوع رویه‌ای قالب‌بندی کنید. برای مثال، با این اعلان‌های داده شده

```
type Func = function(X: Integer): Integer;
var
F: Func;
P: Pointer;
N: Integer;
```

می‌توانید تخصیصات زیر را انجام دهید.

```
F := Func(P); { Assign procedural value in P to F }
Func(P) := F; { Assign procedural value in F to P }
@F := P; { Assign pointer value in P to F }
P := @F; { Assign pointer value in F to P }
```

```
N := F(N); { Call function via F }
N := Func(P)(N); { Call function via P }
```

قالب‌بندی‌های نوع متغیری می‌توانند بوسیله توصیف کننده‌ها دنبال شوند، همان طور که در مثال زیر نشان داده شده است.

```
type
TByteRec = record
Lo, Hi: Byte;
end;
TWordRec = record
Low, High: Word;
end;
PByte = ^Byte;
var
B: Byte;
W: Word;
L: Longint;
P: Pointer;
begin
W := $1234;
B := TByteRec(W).Lo;
TByteRec(W).Hi := 0;
L := $01234567;
W := TWordRec(L).Low;
B := TByteRec(TWordRec(L).Low).Hi;
B := PByte(L)^;
end;
```

در این مثال، *TByteRec* برای دسترسی به بایت‌های رتبه بالا و پایین یک کلمه مورد استفاده قرار گرفته است، و *TWordRec* برای دسترسی به کلمه‌های رتبه بالا و پایین یک عدد صحیح بلند (Longint) مورد استفاده قرار گرفته است. شما می‌توانید از توابع از پیش تعریف شده *Lo* و *Hi* برای همان منظور استفاده کنید، اما یک قالب‌بندی نوع متغیر مزایایی دارد که می‌تواند بر روی سمت چپ یک دستور تخصیص مورد استفاده قرار گیرد. برای اطلاعات بیشتر درباره تبدیل نوع صریح اشاره‌گرها، بخش «اشاره‌گرها و انواع اشاره‌گر» را در فصل۵ ملاحظه نمایید. برای اطلاعات بیشتر درباره تبدیل نوع صریح انواع واسط‌ها و کلاس‌ها، بخش «عملگر **as**» را در فصل۷ و بخش «تبدیلات نوع صریح واسط» را در فصل۱۰ ملاحظه نمایید.

اعلان‌ها و دستورات

گذشته از شرط **uses** (و کلمات رزروشده‌ای مانند **implementation** که حدود بخش‌های مختلف یک یونیت را تعیین می‌کنند)، یک برنامه عموماً از اعلان‌ها و دستوراتی تشکیل شده که در میان بلوک‌ها سازماندهی می‌شوند.

اعلان‌ها

به اسامی متغیرها، ثابت‌ها، نوع‌ها، فیلدها، خاصیت‌ها، روال‌ها، توابع، برنامه‌ها، یونیت‌ها، کتابخانه‌ها و بسته‌ها، شناسه گفته می‌شود. (ثابت‌های عددی مانند 26057 شناسه نیستند.) شناسه‌ها بایستی قبل از این که بتوانید از آنها استفاده کنید، اعلان شوند؛ تنها استثناها تعداد اندکی از نوع‌ها، روتین‌ها و ثوابت از پیش تعریف شده هستند که کامپایلر به طور خودکار آنها را می‌شناسد، متغیر *Result* چنان چه درون یک بلوک تابع رخ دهد، و متغیر *Self* زمانی که در میان یک پیاده‌سازی متد ظاهر شود. اعلان، یک شناسه را تعریف کرده و در جای مقتضی، برایش حافظه تخصیص می‌دهد. برای مثال،

```
var Size: Extended;
```

متغیری به نام *Size* اعلان می‌کند که یک مقدار *Extended* (مقدار حقیقی) را نگه می‌دارد، در حالی که

```
function DoThis(X, Y: string): Integer;
```

یک تابع به نام *DoThis* را تعریف می‌کند که دو رشته را به عنوان آرگومان گرفته و یک عدد صحیح (Integer) را برمی‌گرداند. هر اعلان با یک نقطه ویرگول (;) خاتمه می‌یابد. چنان چه متغیرها، نوع‌ها، ثوابت یا برچسب‌های متعددی را در آن واحد اعلان کنید، بهتر است که واژه‌های کلیدی مقتضی را تنها یک بار بنویسید:

```
var
Size: Extended;
Quantity: Integer;
Description: string;
```

ترکیب نوشتاری و تعیین موقعیت یک اعلان بستگی به نوع شناسه‌ای دارد که تعریف کرده‌اید. عموماً، اعلان‌ها تنها می‌توانند در ابتدای یک بلوک یا ابتدای بخش واسط (**interface**) یا بخش پیاده‌سازی (**implementation**) یک یونیت (بعد از شرط **uses**) ظاهر شوند. قراردادهای خاص مربوط به اعلان متغیرها، ثوابت، نوع‌ها، توابع و ... در همین فصل در عناوین مربوطه‌شان شرح داده شده‌اند.

راهنماهای «تذکر» **platform**، **deprecated** و **library** می‌توانند به هر اعلانی الحاق شوند، به استثنای این که یونیت‌ها نمی‌توانند با **deprecated** اعلان شوند. در مورد اعلان یک تابع یا روال، راهنمای تذکر بایستی با یک نقطه ویرگول (;) از مابقی اعلان جدا گردد. مثال‌ها:

```
procedure SomeOldRoutine; stdcall; deprecated;
var VersionNumber: Real library;
type AppError = class(Exception)
```

```
...
end platform;
```

چنان چه کد منبع در حالت **{$HINTS ON} {$WARNINGS ON}** کامپایل شود، هر ارجاع به یکی از شناسه‌های اعلان شده با یکی از این راهنماهای تذکر، یک تذکر یا هشدار درخور و مناسب تولید می‌کند. از **platform** برای علامت‌گذاری آیتم‌هایی که برای یک محیط عامل به خصوص (مانند ویندوز یا لینوکس) معین شده‌اند، استفاده کنید. از **deprecated** برای نشان دادن این که یک آیتم منسوخ شده یا تنها برای سازگاری با گذشته پشتیبانی می‌شود، استفاده کنید. و از **library** برای پرچم زدن وابستگی‌ها روی یک کتابخانه یا چارچوب اجزای به خصوص (مانند VCL یا CLX)، استفاده کنید.

دستورات

دستورات اعمال الگوریتمی را، میان یک برنامه تعریف می‌کنند. دستورات ساده — مانند تخصیص داده‌ها و فراخوانی روال‌ها— می‌توانند با هم ترکیب شده تا حلقه‌ها، دستورات شرطی و دستورات ساخت‌یافته دیگر را شکل دهند.

دستورات گوناگون واقع در میان یک بلوک و در بخش ارزش‌گذاری (**initialization**) یا اتمام (**finalization**) یک یونیت، با نقطه ویرگول‌ها از هم متمایز می‌شوند.

دستورات ساده

یک دستور ساده خود شامل هیچ دستور دیگری نیست. دستورات ساده شامل تخصیص داده‌ها، فراخوانی روال‌ها و توابع و پرشهای **goto** هستند.

دستورات تخصیص داده

یک دستور تخصیص، شکل زیر را دارد

```
variable := expression
```

جایی که *variable* هر گونه ارجاع متغیری می‌باشد— از جمله یک متغیر، قالب‌بندی نوع متغیر، اشاره‌گر برگشت ارجاع شده یا جزیی از یک متغیر ساخت‌یافته — و *expression* هر گونه عبارت سازگار برای تخصیص می‌باشد. (در میان یک بلوک تابع، *variable* می‌تواند با نام تابع درحال تعریف شدن جایگزین شود. «روال‌ها و توابع» را در فصل ۶ ملاحظه نمایید.) برخی اوقات علامت **:=**

عملگر تخصیص خوانده می‌شود. یک دستور تخصیص مقدار فعلی *variable* را با مقدار *expression* جایگزین می‌کند. برای مثال،

```
I := 3;
```

مقدار 3 را به متغیر *I* تخصیص می‌دهد. متغیر ارجاع شده در سمت چپ عبارت می‌تواند در عبارت سمت راست هم ظاهر شود. برای مثال،

```
I := I + 1;
```

مقدار *I* را به اندازه یک واحد افزایش می‌دهد. نمونه‌های دیگری از تخصیص:

```
X := Y + Z;
Done := (I >= 1) and (I < 100);
Hue1 := [Blue, Succ(C)];
I := Sqr(J) - I * K;
Shortint(MyChar) := 122;
TByteRec(W).Hi := 0;
MyString[I] := 'A';
SomeArray[I + 1] := P^;
TMyObject.SomeProperty := True;
```

فراخوان تابع و روال

فراخوان روال از نام یک روال (با یا بدون قیدها) که با لیستی از پارامترها (اگر لازم باشد) پی گرفته می‌شود، تشکیل می‌شود. چند مثال:

```
PrintHeading;
Transpose(A, N, M);
Find(Smith, William);
Writeln('Hello world!');
DoSomething();
Unit1.SomeProcedure;
TMyObject.SomeMethod(X,Y);
```

با ترکیب نوشتاری بسط یافته فعال (**{$X+}**)، فراخوانی توابع، مانند فراخوان روال‌ها، می‌توانند مانند دستوراتی در حق خودشان، تلقی شوند:

```
MyFunction(X);
```

زمانی که از یک فراخوانی تابع به این شیوه استفاده می‌کنید، مقدار برگشتی‌اش دورانداخته می‌شود. برای اطلاعات بیشتر درباره روال‌ها و توابع، فصل۶، «روال‌ها و توابع» را ملاحظه نمایید.

دستور Goto

دستور **goto**، که فرم زیر را دارد

```
goto label
```

اجرای برنامه را به دستور علامت‌دار شده بوسیله برچسب تعیین شده، منتقل می‌کند. برای علامت‌دار کردن یک دستور، بایستی اول برچسب را اعلان کنید. آنگاه مقدم بر دستور می‌توانید با برچسب و یک دونقطه (**:**) دستور را علامت‌دار کنید:

```
label: statement
```

برچسب‌ها به این صورت اعلان کنید:

```
label label;
```

شما می‌توانید برچسب‌های متعددی را یک باره اعلان کنید:

```
label label1, ..., labeln;
```

یک برچسب می‌تواند هر شناسه معتبر یا هر مقدار عددی بین 0 و 9999 باشد.

اعلان برچسب، دستور علامت‌دار شده و دستور **goto** بایستی به بلوک یکسانی تعلق داشته باشند. (بخش «بلوک‌ها و میدان دید» را در همین فصل ملاحظه نمایید.) از این رو پرش به داخل یا خارج یک روال یا تابع، غیر ممکن است. بیشتر از یک دستور را در یک بلوک با برچسب یکسان علامت‌دار نکنید. برای مثال،

```
label StartHere;
...
StartHere: Beep;
goto StartHere;
```

یک حلقه نامتناهی ایجاد می‌کند که روال *Beep* را مکرراً فراخوانی می‌کند.

معمولاً در برنامه‌نویسی ساخت‌یافته، دستور **goto** دل سرد کننده است. اگر چه این دستور، گاهی به عنوان روشی برای خروج از حلقه‌های تودرتو به کار برده می‌شود، همانند مثال زیر.

```
procedure FindFirstAnswer;
var X, Y, Z, Count: Integer;
label FoundAnAnswer;
begin
Count := SomeConstant;
for X := 1 to Count do
for Y := 1 to Count do
for Z := 1 to Count do
if ... { some condition holds on X, Y, and Z } then
goto FoundAnAnswer;
...    {code to execute if no answer is found }
Exit;
FoundAnAnswer:
...    { code to execute when an answer is found }
end;
```

توجه کنید که در این جا، ما در حال استفاده از **goto** برای پرش به **خارج** از یک حلقه تودرتو هستیم. **هرگز** به **درون** یک حلقه یا دستورات ساخت‌یافته پرش نکنید، زیرا این کار می‌تواند اثرات غیر قابل پیش‌بینی داشته باشد.

دستورات ساخت‌یافته

دستورات ساخت‌یافته از دستورات دیگر ایجاد می‌شوند. چنان چه می‌خواهید دستورات دیگر را به صورت متوالی، مشروط یا مکرر اجرا کنید، از یک دستور ساخت‌یافته استفاده کنید.

- یک دستور مرکب یا **with** به سادگی دنباله ای از دستورات جزیی را اجرا می‌کند.
- یک دستور شرطی— یعنی یک دستور **if** یا **case** — خیلی باشد، حداکثر یکی از اجزایش را، بسته به معیارهای تعیین شده، اجرا می‌کند.
- دستورات حلقه — شامل حلقه‌های **repeat**، **while** و **for** — دنباله ای از دستورات جزیی را به صورت مکرر اجرا می‌کنند.
- گروه خاصی از دستورات — شامل ساختارهای **try...except**, **raise** و **try...finally** — استثناها را ایجاد کرده و اداره می‌کنند. برای اطلاعات بیشتر درباره تولید و اداره استثناها، بخش «استثناها» را در فصل۷ ملاحظه نمایید.

دستورات مرکب

دستور مرکب دنباله‌ای از دستورات دیگر (ساده یا ساخت‌یافته) است که به ترتیبی که نوشته شده‌اند، اجرا می‌شوند. دستور مرکب بوسیله واژه‌های کلیدی **begin** و **end** قلاب شده و دستورات جزیی آن بوسیله نقطه ویرگول‌ها از هم متمایز می‌شوند. برای مثال:

```
begin
Z := X;
X := Y;
Y := Z;
end;
```

آخرین نقطه ویرگول قبل از **end** اختیاری است. بنابراین می‌توانیم کد بالا را به صورت زیر بنویسیم:

```
begin
Z := X;
X := Y;
Y := Z
end;
```

دستورات مرکب در زمینه‌هایی که ترکیب نوشتاری پاسکال شیئی به یک دستور واحد نیاز دارد، حیاتی هستند. علاوه بر بلوک‌های برنامه، تابع و روال، دستورات مرکب نیز در میان دستورات ساخت‌یافته دیگر، مانند شرطی‌ها یا حلقه‌ها، ظاهر می‌شوند. مثال:

```
begin
I := SomeConstant;
while I > 0 do
begin
...
I := I - 1;
end;
end;
```

شما می‌توانید دستور مرکبی بنویسید که تنها حاوی یک دستور جزیی باشد؛ مانند پارانتزها در یک عبارت پیچیده، برخی اوقات **begin** و **end** برای ابهام‌زدایی و بهبود خوانایی به کار می‌روند. همچنین می‌توانید یک دستور مرکب خالی را برای ایجاد یک بلوک که هیچ کاری را انجام نمی‌دهد، به کار ببرید:

```
begin
end;
```

دستور With

دستور **with** یک مختصرنویسی برای ارجاع به فیلدهای یک رکورد یا فیلدها، خاصیت‌ها و متدهای یک شیء است. ترکیب نوشتاری/نحوی دستور **with** به صورت زیر است

```
with obj do statement
```

یا

```
with obj1, ..., objn do statement
```

جایی که *obj* یک ارجاع متغیر مبتنی بر یک شیء یا رکورد است و *statement* هر گونه دستور ساده یا ساخت‌یافته‌ای می‌باشد. در میان *statement* ، تنها با استفاده از شناسه‌ها — بدون قیدها—می‌توانید به فیلدها، خاصیت‌ها و متدهای *obj* ارجاع دهید. برای مثال، برای اعلان‌های داده شده زیر

```
type TDate = record
Day: Integer;
Month: Integer;
Year: Integer;
end;
var OrderDate: TDate;
```

می‌توانید دستور **with** زیر را بنویسید.

```
with OrderDate do
if Month = 12 then
begin
Month := 1;
Year := Year + 1;
end
else
Month := Month + 1;
```

این کد معادل است با:

```
if OrderDate.Month = 12 then
begin
OrderDate.Month := 1;
OrderDate.Year := OrderDate.Year + 1;
end
else
OrderDate.Month := OrderDate.Month + 1;
```

در صورتی که تفسیر *obj* درگیر زیرنویس‌دار کردن آرایه‌ها یا بازگشت ارجاع اشاره‌گرها باشد، قبل از این که *statement* اجرا شود، این اعمال یک مرتبه انجام می‌شوند. این عمل، دستورات **with** را کارا و ضمناً موجز و مختصر می‌سازد. در ضمن به معنای این است که تخصیص‌ها به یک متغیر درمیان *statement* نمی‌تواند بر تفسیر *obj* در طی اجرای فعلی دستور **with**، اثر داشته باشند.

هر ارجاع متغیر یا اسم متد واقع در دستور **with** ، در صورتی که امکان پذیر باشد، به عنوان عضوی از شیء یا رکورد تعیین شده تفسیر می‌شود. در صورتی که در آنجا متغیر یا متد دیگری از همان اسم موجود باشد و شما نیز قصد داشته باشید از دستور **with** به آن دسترسی پیدا کنید، نیازمند این خواهید بود که آن را با یک قید توصیف کنید، مانند مثال زیر:

```
with OrderDate do
begin
Year := Unit1.Year
```

```
...
end;
```

چنان چه اشیاء یا رکوردهای متعددی بعد از **with** ظاهر شوند، سرتاسر دستور مانند یک سری دستورات **with** تودرتو تلقی می‌شود. بنابراین

```
with obj1, obj2, ..., objn do statement
```

معادل است با

```
with obj1 do
with obj2 do
|
with objn do
statement
```

در این حالت، هر ارجاع متغیر یا اسم متد واقع در *statement*، چنان چه امکان‌پذیر باشد، به عنوان عضوی از *objn* تفسیر می‌شود؛ در غیراین صورت، به عنوان یک عضو *objn–1* تفسیر می‌شود؛ و الی آخر. همان قواعد به تفسیر *objs* از خودشان، اعمال می‌شود، به طوری که، برای نمونه، اگر *objn* هم عضوی از *obj1* باشد و هم عضوی از *obj2*، به صورت *obj2.objn* تعبیر می‌شود.

دستور If

دو شکل برای دستور **if** وجود دارد: **if...then** و **if...then...else**. ترکیب نوشتاری دستور **if...then** مانند زیر است

```
if expression then statement
```

جایی که *expression* یک مقدار Boolean را برمی‌گرداند. در صورتی که *expression* **درست** (*True*) باشد، آنگاه *statement* اجرا می‌شود؛ و در غیر این صورت دستور اجرا نمی‌شود. برای مثال،

```
if J <> 0 then Result := I/J;
```

ترکیب نوشتاری/نحوی دستور **if...then...else** به صورت زیر است

```
if expression then statement1 else statement2
```

جایی که *expression* یک مقدار Boolean را برمی‌گرداند. در صورتی که *expression* **درست** (*True*) باشد، آن گاه *statement*1 اجرا می‌شود؛ و در غیر این صورت دستور *statement*2 اجرا می‌شود. برای مثال،

```
if J = 0 then
Exit
else
Result := I/J;
```

شرط‌های **then** و **else** هریک شامل یک دستور هستند، اما این دستور می‌تواند یک دستور ساخت‌یافته نیز باشد. برای مثال،

```
if J <> 0 then
begin
Result := I/J;
Count := Count + 1;
end
else if Count = Last then
Done := True
else
Exit;
```

توجه کنید که **هرگز** یک نقطه ویرگول میان شرط **then** و کلمه **else** وجود ندارد. شما می‌توانید از یک نقطه ویرگول بعد از کلیت دستور **if** برای جدا کردن آن از دستور بعدی در بلوکی که در آن واقع است، استفاده کنید، اما شرط‌های **then** و **else** به هیچ چیزی بیشتر از یک فاصله یا تعویض سطر (carriage return) در میان خود نیاز ندارند.

توجه قرار دادن یک نقطه ویرگول بلافاصله قبل از **else** (در یک دستور **if**) یک اشتباه رایج برنامه نویسی است.

یک مشکل به خصوص در ارتباط با دستورات **if** تودرتو رخ می‌دهد. مشکل از آن جایی ناشی می‌شود که برخی از دستورات **if** شرط **else** دارند درحالی که برخی دیگر ندارند، اما با این حال ترکیب نوشتاری هر دو نوع از دستورات یکی است. در یک سری شرطی‌های تودرتو جایی که شرط‌های **else** کمتری از دستورات **if** وجود دارند، ممکن است واضح نباشد که کدام شرط **else** به کدام یک از **if**ها مقید است. به دستوری با فرم زیر توجه کنید

```
if expression1 then if expression2 then statement1 else statement2;
```

دو روش برای تجزیه این دستور وجود دارد:

```
if expression1 then [ if expression2 then statement1 else statement2 ];
if expression1 then [ if expression2 then statement1 ] else statement2;
```

کامپایلر همواره به روش اولی تجزیه می‌کند. یعنی در واقع، در کد واقعی، دستور

```
if ... { expression1 } then
if ... { expression2 } then
... { statement1 }
else
... { statement2 } ;
```

معادل است با

```
if ... { expression1 } then
begin
if ... { expression2 } then
... { statement1 }
else
... { statement2 }
end;
```

قاعده این است که شرطی‌های تودرتو با شروع از داخلی ترین شرطی تجزیه می‌شوند، با مقید کردن هر **else** به نزدیک‌ترین **if** در دسترس واقع در سمت چپش. برای این که کامپایلر را وادار کنیم تا مثال ما را به روش دومی بخواند، بایستی آن را صریحاً مانند زیر بنویسیم

```
if ... { expression1 } then
begin
if ... { expression2 } then
... { statement1 }
end
else
... { statement2 } ;
```

دستور Case

دستور **case** یک جایگزین خوانا برای شرطی‌های تودرتوی **if** عرضه می‌کند. دستور **case** شکل زیر را دارد

```
case selectorExpression of
   caseList1: statement1;
   ...
   caseListn: statementn;
end
```

جایی که *selectorExpression* هرجور عبارتی از یک نوع ترتیبی بوده (انواع رشته نامعتبر هستند) و هر کدام از *caseList*ها یکی از موارد زیر هستند:

- یک عبارت عددی، ثابت اعلان شده یا عبارت دیگری که کامپایلر بدون اجرای برنامه، می‌تواند آن را ارزیابی کند. بایستی از یک نوع ترتیبی سازگار با *selectorExpression* باشد. بنابراین 'A' , 3*5 + 4 , True , 7 و ('A')Integer همگی می‌توانند به عنوان *caseLists* مورد استفاده قرار گیرند، اما متغیرها و بیشتر فراخوان‌های توابع نمی‌توانند. (تعداد اندکی از توابع توکار مانند *Hi* و *Lo* می‌توانند در یک *caseList* ظاهر شوند. بخش «عبارات ثابت» را در فصل۵ ملاحظه نمایید.)
- یک زیردامنه که فرم *First..Last* را دارد، جایی که *First* و *Last* هردو معیار بالایی را ارضاء می‌کنند و *First* کمتر از یا مساوی با *Last* می‌باشد.
- یک لیست که فرم $item_1, ..., item_n$ را دارد، جایی که *item* یکی از معیارهای بالا را ارضاء می‌کند.

هر مقدار بیان شده بوسیله یک *caseList* واقع در دستور **case** بایستی منحصر به فرد باشد؛ زیردامنه‌ها و لیست‌ها نمی‌توانند همپوشانی داشته باشند. دستور **case** می‌تواند یک شرط **else** نهایی داشته باشد:

```
case selectorExpression of
caseList1: statement1;
...
caseListn: statementn;
else
statements;
end
```

جایی که *statements* دنباله‌ای از دستورات مجزا شده با نقطه ویرگول می‌باشد. زمانی که یک دستور **case** اجرا می‌شود، خیلی باشد حداکثر یکی از $statement_1 ... statement_n$ اجرا می‌شود. هر یک از *caseList*ها که مقدار مساوی با *selectorExpression* داشته باشد، دستوری (*statement*) را که باید اجرا شود، تعیین می‌کند. در صورتی که هیچ کدام از *caseLists* مقدار یکسانی با *selectorExpression* نداشته باشند، آنگاه دستورات شرط **else** (چنان چه چنین شرطی موجود باشد) اجرا می‌شوند. دستور **case**

```
case I of
1..5: Caption := 'Low';
6..9: Caption := 'High';
0, 10..99: Caption := 'Out of range';
else
Caption := '';
end;
```

با شرطی تودرتوی زیر معادل است

```
if I in [1..5] then
Caption := 'Low'
else if I in [6..10] then
Caption := 'High'
else if (I = 0) or (I in [10..99]) then
Caption := 'Out of range'
else
Caption := '';
```

نمونه‌های دیگری از دستور **case**

```
case MyColor of
Red: X := 1;
Green: X := 2;
Blue: X := 3;
Yellow, Orange, Black: X := 0;
end;
```

```
case Selection of
Done: Form1.Close;
Compute: CalculateTotal(UnitCost, Quantity);
else
Beep;
end;
```

حلقه‌های کنترل

حلقه‌ها، با استفاده از یک شرط یا متغیر کنترل برای تعیین زمان توقف اجرای حلقه، به شما اجازه می‌دهند تا یک سری از دستورات را به صورت تکراری انجام دهید. پاسکال شیئی سه نوع حلقه کنترلی دارد: دستور **repeat**، دستور **while**، دستور **for**.

شما می‌توانید از روال‌های استاندارد *Break* و *Continue* برای کنترل گردش کار یک دستور **repeat**، **while** یا **for** استفاده کنید. *Break* در جایی که رخ می‌دهد، به اجرای دستورات خاتمه می‌دهد، در حالی که *Continue* اجرای حلقه را از نمو و تکرار بعدی از سرمی‌گیرد.

دستور Repeat

ترکیب نوشتاری دستور **repeat** به صورت زیر است

```
repeat statement1; ...; statementn; until expression
```

جایی که *expression* یک مقدار Boolean را برمی‌گرداند. (آخرین نقطه ویرگول قبل از **until** اختیاری است.) دستور **repeat**، بعد از هر تکرار با تست کردن *expression*، سلسله دستورات سازنده‌اش را به

صورت پیوسته اجرا می‌کند. چنان چه *expression*، به *True*ارزیابی شود— یعنی *True* را برگرداند— دستور **repeat** خاتمه می‌یابد.

توجه دستور **repeat** حداقل یک بار اجرا می‌شود زیرا *expression* بعد از تکرار اول حلقه و در انتهای آن ارزیابی می‌شود.

نمونه‌هایی از دستور **repeat**

```
repeat
K := I mod J;
I := J;
J := K;
until J = 0;
```

```
repeat
Write('Enter a value (0..9): ');
Readln(I);
until (I >= 0) and (I <= 9);
```

دستور While

دستور **while** مشابه دستور **repeat** است، به استثنای اینکه شرط کنترلی در ابتدای حلقه و قبل از اجرای اولین تکرار آن، ارزیابی می‌شود. از این رو، چنان چه شرط برقرار نباشد—یعنی false باشد— دستورات حلقه هرگز اجرا نمی‌شوند. ترکیب نوشتاری دستور **while** به صورت زیر است

```
while expression do statement
```

جایی که *expression* یک مقدار Boolean را برمی‌گرداند و *statement* می‌تواند یک دستور مرکب باشد. دستور **while** ، قبل از هر تکرار با تست کردن *expression*، دستور تشکیل دهنده‌اش— یعنی *statement* —را به صورت تکراری اجرا می‌کند. تا زمانی که *expression* صحیح باشد— یعنی *True*را برگرداند— اجرا ادامه می‌یابد. نمونه‌هایی از دستور **while**

```
while Data[I] <> X do I := I + 1;
while I > 0 do
begin
if Odd(I) then Z := Z * X;
I := I div 2;
X := Sqr(X);
end;
```

```
while not Eof(InputFile) do
begin
Readln(InputFile, Line);
Process(Line);
```

```
end;
```

دستور For

دستور **for** برخلاف دستور **repeat** یا **while** ، نیازمند این است که تعداد تکرارهایی را که می‌خواهید حلقه انجام دهد، به طور صریح تعیین کنید. ترکیب نوشتاری/نحوی دستور **for** به صورت زیر است

```
for counter := initialValue to finalValue do statement
```

یا

```
for counter := initialValue downto finalValue do statement
```

جایی که

- *counter* یک متغیر محلی (اعلان شده در بلوکی که شامل دستور **for** است) از یک نوع ترتیبی، بدون هرگونه قیدی است .
- *initialValue* و *finalValue* عباراتی هستند که سازگار برای تخصیص با *counter* هستند.
- *statement* یک دستور ساده یا ساخت‌یافته است که مقدار *counter* را تغییر نمی‌دهد.

دستور **for** مقدار *initialValue* را به *counter* تخصیص می‌دهد، سپس، با افزایش یا کاهش *counter* بعد از هر تکرار، *statement* را به صورت تکراری اجرا می‌کند. (ترکیب **for...to** مقدار *counter* را افزایش می‌دهد، در حالی که ترکیب **for...downto** آن را کاهش می‌دهد.) زمانی که *counter* مقدار یکسانی با *finalValue* برگرداند، *statement* یک مرتبه دیگر اجرا شده و دستور **for** خاتمه می‌یابد. به عبارت دیگر، به ازای هر مقدار واقع در دامنه *initialValue* تا *finalValue*، *statement* یک مرتبه اجرا می‌شود. در صورتی که *initialValue* برابر با *finalValue* باشد، *statement* دقیقاً یک مرتبه اجرا می‌شود. اگر *initialValue* در دستور **for...to** بزرگتر از *finalValue*، یا در دستور **for...downto** کمتر از *finalValue* باشد، *statement* هرگز اجرا نمی‌شود. بعد از این که دستور **for** خاتمه یابد، مقدار *counter* تعریف نشده می‌گردد.

از نظر کنترل اجرای حلقه، قبل از این که حلقه شروع شود، عبارات *initialValue* و *finalValue*، صرفاً یک مرتبه ارزیابی می‌شوند. از این رو دستور **for...to** تقریباً، اما نه به طور کامل، با این ساختار **while** معادل است:

```
begin
counter := initialValue;
while counter <= finalValue do
begin
statement;
counter := Succ(counter);
end;
end
```

تفاوت بین این ساختار و دستور **for...to** این است که حلقه **while** قبل از هر تکرار *finalValue* را ارزیابی مجدد می‌کند. در صورتی که *finalValue* یک عبارت پیچیده باشد، این امر می‌تواند منجر به این گردد که کارایی به طور قابل ملاحظه‌ای کندتر شود، در ضمن به معنای این است که تغییرات در مقدار *finalValue* در میان *statement* می‌تواند بر اجرای حلقه اثر داشته باشد. نمونه‌هایی از دستور **for**

```
for I := 2 to 63 do
if Data[I] > Max then
Max := Data[I];
for I := ListBox1.Items.Count - 1 downto 0 do
ListBox1.Items[I] := UpperCase(ListBox1.Items[I]);
for I := 1 to 10 do
for J := 1 to 10 do
begin
X := 0;
for K := 1 to 10 do
X := X + Mat1[I, K] * Mat2[K, J];
Mat[I, J] := X;
end;
for C := Red to Blue do Check(C);
```

بلوک‌ها و دامنه

اعلان‌ها و دستورات در میان بلوک[۱]ها سازماندهی می‌شوند، که این بلوک‌ها فضاهای نام محلی[۲] (یا دامنه[۳]ها) را برای برچسب‌ها و شناسه‌ها تعریف می‌کنند. بلوک‌ها به یک شناسه، مانند یک اسم متغیر، اجازه می‌دهند معانی متفاوتی در بخش‌های مختلفی از یک برنامه داشته باشند. هر بلوک بخشی از اعلان یک برنامه، تابع یا روال است؛ هر اعلان برنامه، تابع یا روال دارای یک بلوک است.

بلوک‌ها

۱ Block
۲ Local Namespace
۳ Scope

هر بلوک از یک سری اعلان که بعد از آنها یک دستور مرکب می‌آید، تشکیل می‌شود. همه اعلان‌ها باید همه با هم در آغاز یک بلوک ظاهر شوند. از این رو قالب یک بلوک به صورت زیر است

```
declarations
begin
  statements
end
```

بخش *declarations* می‌تواند به هر ترتیب دلخواهی، شامل اعلان‌هایی برای متغیرها، ثوابت (از جمله رشته‌های منبع)، نوع‌ها، روال‌ها، توابع و برچسب‌ها باشد. ضمناً در یک بلوک برنامه، بخش اعلان می‌تواند شامل یک یا چند شرط **exports** باشد (فصل ۹، «کتابخانه‌ها و بسته‌ها» را ملاحظه نمایید). برای مثال، در یک اعلان تابع مانند

```
function UpperCase(const S: string): string;
var
  Ch: Char;
  L: Integer;
  Source, Dest: PChar;
begin
  ...
end;
```

اولین سطر اعلان، هدینگ تابع است و همه سطوری که در پی می‌آیند بلوک را می‌سازند. *Ch*, *L*, *Source* و *Dest* متغیرهای محلی هستند؛ اعلان آنها تنها به بلوک تابع *UpperCase* اعمال می‌شوند و— تنها در این بلوک — هرگونه اعلان شناسه‌های مشابه را که ممکن است در بلوک برنامه یا در بخش واسط (**interface**) یا پیاده‌سازی (**implementation**) یک یونیت رخ دهند، باطل می‌کنند.

دامنه (Scope)

یک شناسه، مانند اسم یک متغیر یا تابع، تنها می‌تواند در میان دامنه اعلانش به کار گرفته شود. موقعیت یک اعلان دامنه آن را مشخص می‌کند. یک شناسه اعلان شده در میان اعلان یک برنامه، تابع یا روال، دامنه محدود به بلوکی که در آن اعلان شده است، دارد. یک شناسه اعلان شده در بخش واسط (**interface**) یک یونیت دامنه‌ای دارد که شامل هر یونیت یا برنامه دیگری که از این یونیت استفاده می‌کند—جایی که اعلان رخ می‌دهد— می‌شود. شناسه‌ها با دامنه محدودتر— به ویژه شناسه‌های اعلان شده در میان توابع و روال‌ها— برخی اوقات شناسه‌های محلی[۱] خوانده می‌شوند، درحالی که

[۱] Local

شناسه‌هایی با دامنه وسیع‌تر، سراسری[۱] خوانده می‌شوند. قواعدی که دامنه شناسه را مشخص می‌کنند در زیر جمع‌بندی شده‌اند.

درصورتی که شناسه در . . . . اعلان شده باشد	دامنه‌اش . . . . توسعه می‌یابد
اعلان یک برنامه، تابع یا روال	از نقطه‌ای که در آن جا اعلان شده تا انتهای بلوک جاری، از جمله همه بلوک‌های محصور شده در میان آن دامنه
بخش واسط (**interface**) یک یونیت	از نقطه‌ای که در آن جا اعلان شده تا انتهای یونیت، و هر یونیت یا برنامه دیگری که از این یونیت استفاده می‌کنند.
بخش پیاده‌سازی (**implementation**) یک یونیت، اما نه در میان بلوک هر تابع یا روالی	از نقطه‌ای که در آن جا اعلان شده تا انتهای یونیت . شناسه برای هر تابع یا روالی که در این یونیت اعلان شده، از جمله بخش‌های ارزش‌گذاری (**initialization**) و اتمام (**finalization**)، چنان چه حاضر باشند، در دسترس است.
تعریف یک نوع رکورد (یعنی، شناسه اسم یک فیلد واقع در رکورد است)	از نقطه اعلانش تا انتهای تعریف نوع رکورد.
تعریف یک کلاس (یعنی، شناسه اسم یک خاصیت فیلد داده یا متد واقع در کلاس است)	از نقطه اعلانش تا انتهای تعریف نوع کلاس، و همچنین فرزندان کلاس و بلوک‌های همه متدهای کلاس و فرزندانش را نیز دربرمی‌گیرد.

تعارضات نام گذاری

[۱] Global

هنگامی‌که یک بلوک، بلوک دیگری را در برمی‌گیرد، بلوک اولی بلوک خارجی خوانده می‌شود و بلوک دومی، بلوک درونی خوانده می‌شود. اگر یک شناسه که در بلوکی بیرونی اعلان شده باشد در یک بلوک درونی اعلان مجدد شود، اعلان درونی، اعلان بیرونی را باطل کرده و منظور شناسه را برای مدت دوام بلوک داخلی مشخص می‌کند. برای مثال، چنان چه یک متغیر با نام *MaxValue* را در بخش واسط (**interface**) یک یونیت اعلان کنید، و سپس متغیر دیگری را با همین نام در یک اعلان تابع واقع در میان این یونیت اعلان کنید، هر گونه رخداد قیددار نشده *MaxValue* در بلوک تابع بوسیله دومی— یعنی اعلان محلی— کنترل می‌شود. به همین نحو، یک تابع اعلان شده در میان تابعی دیگر یک دامنه جدید ایجاد می‌کند که در آن شناسه‌های استفاده شده بوسیله تابع خارجی می‌توانند به طور محلی اعلان مجدد شوند.

استفاده از یونیت‌های متعدد قدری تعریف دامنه را پیچیده تر می‌کند. هر یونیت لیست شده در شرط **uses** یک دامنه جدید تحمیل می‌کند که یونیت‌های باقیمانده استفاده شده و برنامه یا یونیت حاوی شرط **uses** را دربرمی‌گیرد. اولین یونیت واقع در شرط **uses** بیانگر بیرونی‌ترین دامنه است و هر یونیت بعدی بیانگر یک دامنه جدید در میان یونیت قبلی است. اگر دو یا چند یونیت شناسه یکسانی را در بخش واسط (**interface**) خود اعلان کنند، یک ارجاع بدون قید به این شناسه، اعلان واقع در درونی‌ترین دامنه را انتخاب می‌کند— یعنی، در یونیتی که در آن خود ارجاع ظاهر می‌شود، یا، اگر آن یونیت این شناسه را اعلان نکرده باشد، در آخرین یونیت واقع در شرط **uses** که شناسه را اعلان کرده باشد.

یونیت *System* به طور خودکار به وسیله هر یونیت یا برنامه مورد استفاده قرار می‌گیرد. اعلان‌های واقع در *System*، همراه با نوع‌ها، روتین‌ها و ثوابت از پیش تعریف شده‌ای که کامپایلر آنها را به طور خودکار می‌شناسد، همیشه بیرونی‌ترین دامنه را دارند.

می‌توانید با استفاده از یک شناسه قیددار یا یک دستور **with**، قواعد دامنه را باطل کرده و یک اعلان داخلی را دور بزنید. (برای جزییات بیشتر، بخش‌های «شناسه‌های توصیف شده» و «دستور **with**» را در همین فصل ملاحظه نمایید.)

فصل ۵

انواع داده، متغیرها و ثوابت

نوع[1] در اصل یک اسم برای یک نوع از داده است. زمانی که یک متغیر را تعریف می‌کنید باید نوع آن را مشخص کنید، که این نوع مجموعه‌ای از مقادیر را که متغیر می‌تواند نگه دارد و عملیات‌هایی را که می‌توان روی آن انجام داد، مشخص می‌کند. هر عبارت داده‌ای از یک نوع به خصوص را برمی‌گرداند، همان طور که هر تابع این کار را انجام می‌دهد. اغلب توابع و روال‌ها به پارامترهایی از نوعهایی خاص نیاز دارند.

پاسکال شیئی یک زبان «شدیداً نوع گرا» می‌باشد، یعنی این که انواع داده متنوع و وسیعی را تشخیص داده و هیچ‌گاه به شما اجازه نمی‌دهد تا یکی از انواع داده را با دیگری جایگزین کنید. این امر معمولاً سودمند است، زیرا به کامپایلر اجازه می‌دهد که با داده‌ها هوشمندانه رفتار کرده و — با ممانعت از بروز خطاهایی که در زمان اجرا به سختی تشخیص داده می‌شوند — کد شما را به طور کامل تصدیق کند. گرچه چنان چه به انعطاف بیشتری نیاز داشته باشید، مکانیزم‌هایی برای گیرانداختن طبقه‌بندی نوع

---

[1] Type

قوی وجود دارند. این مکانیزم‌ها شامل قالب‌بندی نوع (تبدیل نوع صریح)، اشاره‌گرها، واریانت‌ها، بخش‌های واریانت در رکوردها و آدرس دهی مطلق متغیرها می‌باشد.

مباحثی درباره نوع‌ها

روش‌های متعددی برای فهرست‌بندی انواع داده در پاسکال شیئی وجود دارد:

- برخی از نوع‌ها از پیش تعریف شده (یا توکار) هستند؛ بدون نیاز به هیچ اعلانی، کامپایلر آنها را به طور خودکار شناسایی می‌کند. تقریباً همه نوع‌های مستند شده در این مرجع زبان از پیش تعریف شده هستند. نوع‌های دیگر از طریق اعلان ایجاد می‌شوند؛ که این نوع‌ها شامل نوع‌های تعریف شده کاربر و نوع‌های تعریف شده در کتابخانه‌های محصول می‌باشند.
- نوع‌ها می‌توانند یا به صورت بنیادی یا عمومی طبقه‌بندی شوند. دامنه و فرمت یک نوع بنیادی — قطع نظر از CPU و سیستم عامل متضمن— در همه پیاده‌سازی‌های پاسکال شیئی یکسان هستند. دامنه و فرمت یک نوع عمومی خاص پلت‌فرم هستند و در میان پیاده‌سازی‌های متفاوت تغییر می‌کنند. اغلب نوع‌های از پیش تعریف شده، بنیادی هستند، اما تنی چند از انواع عدد صحیح، کاراکتر، رشته و اشاره‌گر، عمومی هستند. این ایده جالبی است که هر زمان امکان‌پذیر باشد از انواع عمومی‌استفاده کرد، زیرا عملکرد و انتقال‌پذیری بهینه‌ای را عرضه می‌کنند. اگرچه تغییرات در فرمت ذخیره‌سازی از یک پیاده‌سازی یک نوع عمومی به بعدی می‌تواند منجر به مشکلات سازگاری گردد— برای مثال، زمانی که در حال جریان دادن داده به یک فایل باشید.
- نوع‌ها می‌توانند به صورت ساده، رشته، ساخت یافته، اشاره‌گر، رویه ای یا واریانت طبقه بندی شوند. اضافه بر این، شناسه‌های نوع خودشان می‌توانند به عنوان دارایی برای یک «نوع» خاص مورد نظر قرار گیرند زیرا آنها می‌توانند به عنوان پارامتر برای توابع به خصوصی ارسال شوند (مانند توابع *High*، *Low* و *SizeOf*).

شمای کلی زیر طبقه بندی انواع داده در پاسکال شیئی را نشان می‌دهد.

simple	**ساده**
ordinal	ترتیبی
integer	صحیح
character	کاراکتر
Boolean	بولی
enumerated	شمارشی
subrange	زیردامنه
real	حقیقی
string	رشته
structured	ساخت یافته
set	مجموعه
array	آرایه
record	رکورد
file	فایل
class	کلاس
class reference	ارجاع کلاس
interface	واسط (رابط)
pointer	اشاره‌گر
procedural	رویه‌ای
variant	واریانت
type identifier	شناسه نوع

تابع استاندارد *SizeOf* روی همه متغیرها و شناسه‌های نوع عمل می‌کند. این تابع یک عدد صحیح برمی‌گرداند که بیانگر اندازه حافظه استفاده شده (به بایت) برای ذخیره داده‌ای از نوع تعیین شده است. برای مثال، SizeOf(Longint) مقدار ۴ را برمی‌گرداند، زیرا یک متغیر *Longint* چهار بایت از حافظه را مصرف می‌کند.

اعلانات نوع در بخش‌هایی که می‌آیند کاملاً تشریح می‌شوند. برای اطلاعات کلی درباره اعلان‌های نوع، «اعلان نوع‌ها» را در همین فصل ملاحظه نمایید.

انواع ساده

انواع ساده، که شامل انواع ترتیبی و انواع حقیقی هستند، مجموعه‌های مرتبی از مقادیر را تعریف می‌کنند.

انواع ترتیبی (Ordinal types)

انواع ترتیبی شامل نوع‌های شمارشی، بولی، کاراکتر، زیردامنه و صحیح می‌باشند. یک نوع ترتیبی مجموعه مرتبی از مقادیر را تعریف می‌کند که در آن، به استثنای مقدار اولی، هر مقدار یک سلف و ماقبل منحصر به فردی دارد و هر مقدار، به استثنای مقدار آخری، یک خلف و مابعد منحصر به فردی دارد. اضافه بر این، هر مقدار یک رتبه دارد که ترتیب نوع را مشخص می‌کند. در اغلب موارد، در صورتی که یک مقدار رتبه *n* داشته باشد، سلف آن رتبه *n*–1 و خلف آن رتبه *n*+1 را دارد.

- برای انواع صحیح، رتبه یک مقدار، خود مقدار می‌باشد.
- نوع‌های زیردامنه رتبه‌های نوع پایه ای خود را نگه می‌دارند.
- برای نوع‌های ترتیبی دیگر، به طور پیش فرض مقدار اولی رتبه صفر را دارد، مقدار بعدی رتبه یک و الی آخر. اعلان یک مقدار شمارشی به طور صریح می‌تواند این پیش فرض را باطل کند.

توابع از پیش تعریف شده متعددی روی مقادیر ترتیبی و شناسه‌های نوع عمل می‌کنند. مهمترین آنها در زیر جمع‌بندی شده‌اند.

تابع	پارامتر	مقدار برگشتی	توضیحات
Ord	عبارت ترتیبی (ordinal)	رتبه مقدار عبارت	آرگومان‌های *Int64* را نمی‌پذیرد
Pred	عبارت ترتیبی (ordinal)	سلف مقدار عبارت	بر روی خاصیت‌هایی که روال **write** دارند استفاده نمی‌شود.

Succ	عبارت ترتیبی (ordinal)	خلف مقدار عبارت	بر روی خاصیت‌هایی که روال **write** دارند استفاده نمی‌شود.
High	شناسه نوع ترتیبی (ordinal) یا متغیری از نوع ترتیبی	بالاترین مقدار در نوع	بر روی نوع‌های رشته کوتاه و آرایه‌ها نیز عمل می‌کند.
Low	شناسه نوع ترتیبی (ordinal) یا متغیری از نوع ترتیبی	پایین‌ترین مقدار در نوع	بر روی نوع‌های رشته کوتاه و آرایه‌ها نیز عمل می‌کند.

برای مثال، High(Byte) مقدار 255 را برمی‌گرداند زیرا بالاترین مقدار از نوع *Byte* برابر 255 است، و Succ(2) مقدار 3 را برمی‌گرداند زیرا 3 خلف 2 است.

روال‌های استاندارد *Inc* و *Dec* مقدار یک نوع ترتیبی را افزایش و کاهش می‌دهند. برای مثال، Inc(I) معادل است با I := Succ(I) و چنان چه *I* یک متغیر صحیح باشد، معادل است با I := I + 1.

انواع صحیح (Integer)

یک نوع صحیح بیانگر زیرمجموعه‌ای از اعداد کامل می‌باشد. *Integer* و *Cardinal* انواع صحیح عمومی هستند؛ از این نوع‌ها، هر زمان که امکان‌پذیر باشد استفاده کنید، زیرا بهترین کارایی را برای CPU و سیستم عامل متضمن نتیجه می‌دهند. جدول زیردامنه‌ها و فرمت ذخیره‌سازی را برای کامپایلر ۳۲-بیت پاسکال شیئی، نشان می‌دهد.

Table 5.1 انواع صحیح عمومی برای پیاده‌سازی‌های ۳۲ بیتی پاسکال شیئی

نوع	دامنه	فرمت
Integer	–2147483648..2147483647	signed 32-bit
Cardinal	0..4294967295	unsigned 32-bit

انواع بنیادین صحیح شامل *Shortint*, *Smallint*, *Longint*, *Int64*, *Byte*, *Word* و *Longword* هستند.

Table 5.2 انواع صحیح بنیادین

نوع	دامنه	فرمت
Shortint	–128..127	signed 8-bit
Smallint	–32768..32767	signed 16-bit

Longint	–2147483648..2147483647	signed 32-bit
Int64	$-2^{63}..2^{63}-1$	signed 64-bit
Byte	0..255	unsigned 8-bit
Word	0..65535	unsigned 16-bit
Longword	0..4294967295	unsigned 32-bit

به طور کلی، عملیات‌های ریاضی روی اعداد صحیح مقداری از نوع *Integer* —که در پیاده‌سازی فعلی‌اش، با *Longint* ۳۲–بیت معادل است— برمی‌گردانند. عملیات‌ها مقداری از نوع *Int64* را تنها زمانی برمی‌گردانند که روی یک عملوند *Int64* اجرا شوند. از این رو کد زیر نتایج نادرستی تولید می‌کند.

```
var
I: Integer;
J: Int64;
...
I := High(Integer);
J := I + 1;
```

برای گرفتن یک مقدار برگشتی *Int64* در یک چنین وضعیتی، *I* را به صورت *Int64* قالب‌بندی کنید:

```
...
J := Int64(I) + 1;
```

برای آگاهی از اطلاعات بیشتر بخش «عملگرهای محاسباتی» را در فصل ۴ ملاحظه کنید.

توجه اغلب روتین‌های استاندارد که آرگومان‌های صحیح می‌گیرند، مقادیر *Int64* را به ۳۲–بیت می‌رسانند. اگرچه، روتین‌های *IntToStr*, *Succ*, *Pred*, *Inc*, *Dec*, *High*, *Low* و *IntToHex* به طور کامل آرگومان‌های *Int64* را پشتیبانی می‌کنند. در ضمن، توابع *Round*, *Trunc*, *StrToInt64* و *StrToInt64Def* مقادیر *Int64* را برمی‌گردانند. تعداد اندکی روتین —از جمله *Ord* — به هیچ وجه نمی‌توانند مقادیر *Int64* را بپذیرند.

چنان چه آخرین مقدار از یک نوع صحیح را نمو دهید یا اولین مقدارش را به اندازه یک واحد کاهش دهید، نتیجه در پیرامون آغاز یا انتهای دامنه می‌پیچد. برای مثال، نوع *Shortint* دامنه 128..127– را داراست؛ از این رو، بعد از اجرای کد

```
var I: Shortint;
...
I := High(Shortint);
I := I + 1;
```

مقدار *I* برابر 128– است. به هر حال، چنان چه وارسی دامنه کامپایلر[۱] فعال باشد، این کد یک خطای زمان اجرا تولید خواهد کرد.

انواع کاراکتر

قبل از فراگیری مطالب این بخش به یاد داشته باشید که هر بایت برابر ۸ بیت است و هر کلمه برابر ۲ بایت یا ۱۶ بیت است.

1 Byte = 8 Bit
1 Word = 2 Byte = 16 Bit

AnsiChar و *WideChar* انواع بنیادین کاراکتر هستند. مقادیر *AnsiChar* کاراکترهایی به اندازه ۱ بایت (۸–بیت) هستند که مطابق با مجموعه کاراکتر منطقه‌ای، که شاید مولتی بایت باشد، مرتب شده اند. *AnsiChar* دراصل بعد از مجموعه کاراکتر ANSI مدل سازی شده بود (مثل نامش) اما اکنون برای ارجاع به مجموعه کاراکتر منطقه جاری منتشر شده است.

کاراکترهای *WideChar* برای نمایش هر کاراکتر بیش از یک بایت را به کار می‌برند. در پیاده‌سازی جاری، *WideChar* کاراکترهایی به اندازه یک کلمه (۱۶– بیت) هستند که برطبق مجموعه کاراکتر یونیکد مرتب شده اند (توجه کنید که در پیاده‌سازی‌های بعدی می‌تواند بلندتر شود). ۲۵۶ کاراکتر اول یونیکد با کاراکترهای ANSI منطبق هستند.

Char نوع عمومی کاراکتر می‌باشد، که معادل با *AnsiChar* است. از آن جا که پیاده‌سازی *Char* در معرض تغییر قرار دارد، این ایده خوبی است که در زمان نوشتن برنامه‌هایی که ممکن است نیازمند اداره کردن کاراکترها از انواع مختلف باشند، از تابع استاندارد *SizeOf* به جای یک متن با کد مشکل، استفاده شود.

یک ثابت رشته‌ای با طول یک، مانند 'A'، می‌تواند یک مقدار کاراکتر را مشخص کند. تابع از پیش تعریف شده *Chr* یک مقدار کاراکتر برای هر عدد صحیح در دامنه *AnsiChar* یا *WideChar* برمی‌گرداند؛ برای مثال، Chr(65) حرف *A* را برمی‌گرداند.

---

[۱] Compiler range-checking

مقادیر کاراکتر، مانند اعداد صحیح، چنان چه به دور از ابتدا یا انتهای دامنه‌شان نمو یا کاهش یابند، در حول ابتدا و انتهای دامنه می‌پیچند (مگر این که وارسی دامنه فعال باشد). برای مثال، بعد از اجرای کد

```
var
Letter: Char;
I: Integer;
begin
Letter := High(Letter);
for I := 1 to 66 do
Inc(Letter);
end;
```

Letter مقدار A (ASCII 65) را دارد.

برای آگاهی از اطلاعات بیشتر درباره کاراکترهای یونیکد بخش‌های «بحثی درباره مجموعه کاراکترهای بسط یافته» و «کار کردن با رشته‌های منتهی به تهی» را در همین فصل ملاحظه نمایید.

انواع بولی (Boolean)

چهار نوع بولی از پیش تعریف شده، *Boolean*, *ByteBool*, *WordBool* و *LongBool* هستند. نوع ارجح بر بقیه *Boolean* است. سه نوع دیگر برای سازگاری با زبان‌ها و کتابخانه‌های سیستم عامل‌های دیگر به وجود آمده‌اند.

متغیر *Boolean* یک بایت از حافظه را اشغال می‌کند، در ضمن متغیر *ByteBool* هم یک بایت را اشغال می‌کند، متغیر *WordBool* دو بایت (معادل یک کلمه) را اشغال می‌کند، و یک متغیر *LongBool* چهار بایت (معادل دو کلمه) را اشغال می‌کند.

مقادیر بولی به وسیله ثوابت از پیش تعریف شده *True* و *False* تفکیک می‌شوند. روابط زیر برای نوع‌های بولی برقرار می‌باشند.

Boolean	ByteBool, WordBool, LongBool
False < *True*	*False* <> *True*
Ord(*False*) = 0	*Ord*(*False*) = 0
Ord(*True*) = 1	*Ord*(*True*) <> 0
Succ(*False*) = *True*	*Succ*(*False*) = *True*
Pred(*True*) = *False*	*Pred*(*False*) = *True*

مقداری از نوع *ByteBool* ، *LongBool* یا *WordBool*، در صورتی که رتبه اش غیر صفر باشد، به عنوان *True* در نظر گرفته می‌شود. در صورتی که یک چنین مقداری در یک متن، جایی که یک *Boolean* مورد انتظار است، ظاهر شود کامپایلر به طور خودکار هر مقداری با رتبه غیر صفر را به *True* تبدیل می‌کند.

توضیحات بالا به رتبه مقادیر بولی برمی‌گردد — نه به خود مقدارها. در پاسکال شیئی، عبارات بولی نمی‌توانند با اعداد صحیح یا حقیقی یکسان در نظر گرفته شوند. از این رو، اگر *X* یک متغیر صحیح باشد، دستور

```
if X then ...;
```

یک خطای کامپایل تولید می‌کند. قالب‌بندی متغیر به یک نوع بولی غیرقابل اعتماد است، اما هر کدام از چاره‌های زیر به خوبی جواب می‌دهند.

```
if X <> 0 then ...; { use longer expression that returns Boolean value }
var OK: Boolean { use Boolean variable }
...
if X <> 0 then OK := True;
if OK then ...;
```

انواع شمارشی

یک نوع شمارشی به سادگی مجموعه مرتبی از مقادیر را با لیست کردن شناسه‌هایی که این مقادیر را مشخص می‌کنند، تعریف می‌کند. مقادیر معنای ذاتی ندارند. برای اعلان یک نوع شمارشی، از ترکیب نوشتاری زیر استفاده کنید

```
type typeName = (val1, ..., valn)
```

جایی که *typeName* و هر یک از *val* ها شناسه‌های معتبری هستند. برای مثال، اعلان

```
type Suit = (Club, Diamond, Heart, Spade);
```

یک نوع شمارشی با نام *Suit* تعریف می‌کند که مقادیر امکان‌پذیر آن *Club, Diamond, Heart* و *Spade* هستند، جایی که Ord(Club) مقدار صفر، Ord(Diamond) مقدار یک و . . . را برمی‌گرداند.

هنگامی که یک نوع شمارشی را اعلان می‌کنید، هر *val* را طوری اعلان می‌کنید که یک ثابت از نوع *typeName* باشد. در صورتی که در بین همان دامنه شناسه‌های *val* برای اهداف دیگری به کار گرفته شوند، تعارضات نام‌گذاری بروز می‌کند. برای مثال، تصور کنید که نوع زیر را اعلان کرده باشید

```
type TSound = (Click, Clack, Clock);
```

متأسفانه، *Click* نام یک متد تعریف شده برای *TControl* و همه اشیایی که در VCL و/یا CLX از آن مشتق می‌شوند، است. بنابراین اگر در حال نوشتن برنامه‌ای هستید و یک گرداننده رویداد مانند زیر ایجاد می‌کنید

```
procedure TForm1.DBGrid1Enter(Sender: TObject);
var Thing: TSound;
begin
...
Thing := Click;
...
end;
```

یک خطای کامپایل شدن را دریافت خواهید کرد؛ کامپایلر *Click* را میان دامنه روال به عنوان یک ارجاع به متد *Click* متعلق به *TForm* تعبیر می‌کند. برای حل این مشکل می‌توانید شناسه‌ها را قیددار نمایید، بنابراین چنان چه *TSound* در *MyUnit* اعلان شود، می‌توانید از دستور زیر استفاده کنید

```
Thing := MyUnit.Click;
```

اگرچه، راه حل بهتر این است که نام‌های ثوابت را به گونه‌ای انتخاب کنید که احتمال ناسازگاری با شناسه‌های دیگر را نداشته باشند. چند نمونه:

```
type
TSound = (tsClick, tsClack, tsClock);
TMyColor = (mcRed, mcBlue, mcGreen, mcYellow, mcOrange);
Answer = (ansYes, ansNo, ansMaybe);
```

می‌توانید به طور مستقیم از ساختار (*val*1, ..., *val*n) در اعلان متغیرها استفاده کنید، انگار که نام یک نوع بوده است:

```
var MyCard: (Club, Diamond, Heart, Spade);
```

اما اگر *MyCard* را با این روش اعلان کنید، در میان همان دامنه نمی‌توانید متغیر دیگری را، با استفاده از این شناسه‌های ثابت اعلان کنید. از این رو

```
var Card1: (Club, Diamond, Heart, Spade);
var Card2: (Club, Diamond, Heart, Spade);
```

یک خطای کامپایل تولید می‌کند. اما

```
var Card1, Card2: (Club, Diamond, Heart, Spade);
```

به درستی کامپایل می‌شود، همان طور که کد زیر هم این گونه است

```
type Suit = (Club, Diamond, Heart, Spade);
var
Card1: Suit;
Card2: Suit;
```

انواع شمارشی با رتبه تخصیص داده شده به طورصریح

به طور پیش فرض، رتبه‌های مقادیر شمارشی از صفر شروع شده و همان توالی را که شناسه‌های آنها در اعلان نوع لیست شده‌اند، پی می‌گیرند. شما می‌توانید این روند پیش فرض را با تخصیص رتبه‌ها به طور صریح به برخی یا همه مقادیر واقع در اعلان باطل کنید.

برای تخصیص رتبه به یک مقدار، شناسه آن را با *constantExpression*= پی بگیرید، جایی که *constantExpression* یک عبارت ثابت است که به یک عدد صحیح ارزیابی می‌شود. (بخش «عبارات ثابت» را در همین فصل ببینید) برای مثال،

```
type Size = (Small = 5, Medium = 10, Large = Small + Medium);
```

یک نوع با نام *Size* تعریف می‌کند که مقادیر امکان‌پذیر آن *Small*، *Medium* و *Large* هستند، جایی که Ord(Small)، مقدار 5 را برمی‌گرداند، Ord(Medium) مقدار 10 را برمی‌گرداند و Ord(Large) مقدار 15 را برمی‌گرداند.

یک نوع شمارشی در واقع، یک زیر دامنه است که پایین‌ترین و بالاترین مقادیرش با کمترین و بالاترین رتبه ثابت‌های واقع در اعلان، منطبق هستند.

در مثال بالا، نوع *Size* دارای 11 مقدار امکان‌پذیر است که رتبه‌هایشان از 5 تا 15 تغییر می‌کنند. (از این رو **array**[Size] **of** Char بیانگر آرایه‌ای از 11 کاراکتر است.) تنها سه تا از این مقادیر اسم دارند، اما بقیه می‌توانند از طریق قالب بندی و روتین‌هایی مانند *Inc*, *Succ*, *Pred* و *Dec* قابل دسترس باشند. در مثال زیر، مقادیر «بی‌نام» واقع در دامنه *Size* به متغیر *X* تخصیص داده می‌شوند.

```
var X: Size;
X := Small; // Ord(X) = 5
X := Size(6); // Ord(X) = 6
```

```
Inc(X); // Ord(X) = 7
```

هر مقداری که به طور صریح یک رتبه تخصیص داده شده نداشته باشد، رتبه ای که یک واحد بزرگتر از مقدار قبلی در لیست است، خواهد داشت. چنان چه برای نخستین مقدار رتبه‌ای تخصیص داده نشده باشد، رتبه‌اش صفر خواهد بود. از این رو با اعلان داده شده زیر

```
type SomeEnum = (e1, e2, e3 = 1);
```

SomeEnum تنها دو مقدار امکان‌پذیر دارد: Ord(e1) مقدار صفر را برمی‌گرداند، Ord(e2) مقدار ۱ را برمی‌گرداند و Ord(e3) هم مقدار ۱ را برمی‌گرداند؛ زیرا از آن جایی که *e2* و *e3* رتبه یکسانی دارند، بیانگر مقدار یکسانی هستند.

انواع زیردامنه (Subrange)

یک نوع زیردامنه (subrange) بیانگر زیردامنه ای از مقادیر واقع در نوع ترتیبی دیگری است (که نوع مبنا خوانده می‌شود). هر ساختار به فرم *Low..High*، جایی که *Low* و *High* عبارات ثابتی از نوع ترتیبی یکسانی هستند و *Low* کمتر از *High* است، یک نوع زیر دامنه را مشخص می‌کند که شامل همه مقادیر واقع در میان *Low* و *High* می‌باشد. برای مثال، چنان چه نوع شمارشی زیر را اعلان کنید

```
type TColors = (Red, Blue, Green, Yellow, Orange, Purple, White, Black);
```

در این صورت می‌توانید یک نوع زیردامنه (subrange) مانند زیر تعریف کنید

```
type TMyColors = Green..White;
```

در اینجا *TMyColors* شامل مقادیر *Green, Yellow, Orange, Purple* و *White* می‌باشد.

برای تعریف انواع زیردامنه، می‌توانید از ثوابت عددی و کاراکترها (ثوابت رشته ای با طول ۱) استفاده کنید.

```
type
SomeNumbers = -128..127;
Caps = 'A'..'Z';
```

زمانی که از ثوابت عددی یا کاراکتر برای تعریف یک زیردامنه استفاده می‌کنید، نوع مبنا، نوع کوچک‌ترین عدد صحیح یا کاراکتری است که در دامنه تعیین شده وجود دارد.

ساختار *Low..High* خودش مانند یک اسم نوع عمل می‌کند، بنابراین می‌توانید آن را به طور مستقیم در اعلان‌های متغیر به کار ببرید. برای مثال،

```
var SomeNum: 1..500;
```

متغیر صحیحی تعریف می‌کند که مقدار آن می‌تواند در هر جایی از دامنه 1 تا 500 باشد.

رتبه هر مقدار در یک زیردامنه از نوع مبنا نگه داشته می‌شود. (در اولین مثال واقع در مراحل قبلی، چنان چه *Color* متغیری باشد که مقدار *Green* را نگه دارد، Ord(Color) مقدار ۲ را برمی‌گرداند علیرغم این که آیا *Color* از نوع *TColors* باشد یا از نوع *TMyColors*.) مقادیر حول ابتدا یا انتهای یک زیردامنه نمی‌پیچند، حتی اگر مبنا یک عدد صحیح یا نوع کاراکتر باشد؛ نمو یا کاهش خارج از مرز یک زیر دامنه، در اصل، مقادیر را به نوع مبنا تبدیل می‌کند. از این رو، در حالی که

```
type Percentile = 0..99;
var I: Percentile;
...
I := 100;
```

یک خطا تولید می‌کند،

```
...
I := 99;
Inc(I);
```

مقدار 100 را به *I* تخصیص می‌دهد (مگر این که حالت وارسی دامنه کامپایلر فعال باشد).

استفاده از عبارات ثابت در تعاریف زیردامنه سبب یک اشکال نحوی می‌شود. در هر اعلان نوع، زمانی که اولین کاراکتر معنادار بعد از = یک پارانتز چپ (— — باشد، کامپایلر فرض می‌کند که یک نوع شمارشی در حال تعریف شدن است. از این رو کد زیر

```
const
X = 50;
Y = 10;
type
Scale = (X - Y) * 2..(X + Y) * 2;
```

یک خطا تولید می‌کند. برای حل این مشکل، اعلان نوع را بازنویسی می‌کنیم تا از پارانتز مقدم پرهیز شود:

```
type
Scale = 2 * (X - Y)..(X + Y) * 2;
```

انواع حقیقی (Real)

نوع حقیقی مجموعه‌ای از اعداد را تعریف می‌کند که می‌تواند با نمادگذاری ممیز شناور نمایش داده شود. جدول زیر دامنه‌ها و فرمت‌های ذخیره‌سازی را برای انواع حقیقی اصلی نمایش می‌دهد.

Table 5.3 انواع حقیقی بنیادین

نوع	دامنه	ارقام بامعنی	اندازه به بایت
Real48	$2.9\times10^{-39}..1.7\times10^{38}$	11–12	6
Single	$1.5\times10^{-45}..3.4\times10^{38}$	7–8	4
Double	$5.0\times10^{-324}..1.7\times10^{308}$	15–16	8
Extended	$3.6\times10^{-4951}..1.1\times10^{4932}$	19–20	10
Comp	$-2^{63}+1..2^{63}-1$	19–20	8
Currency	–922337203685477.5808.. 922337203685477.5807	19–20	8

نوع عمومی *Real*، در پیاده‌سازی فعلی‌اش، معادل با *Double* است.

Table 5.4 انواع حقیقی عمومی

نوع	دامنه	ارقام بامعنی	اندازه به بایت
Real	$5.0\times10^{-324}..1.7\times10^{308}$	15–16	8

توجه نوع *Real48* شش بایتی در نسخه‌های اولیه پاسکال شیئی *Real* خوانده می‌شد. اگردر حال کامپایل کردن کدی هستید که از نوع قدیمی‌تر، یعنی نوع *Real* شش بایتی، استفاده می‌کند، شاید بخواهید آن را به *Real48* تغییر دهید. در ضمن می‌توانید از راهنمای کامپایلر **{$REALCOMPATIBILITY ON}** استفاده کنید تا *Real* را به نوع شش بایتی برگردانید.

نکات زیر به انواع حقیقی بنیادی اعمال می‌شوند.

- *Real48* برای سازگاری با گذشته پشتیبانی می‌شود. از آن جا که فرمت نگه داری آن برای خانواده Intel CPU بومی نیست، کارایی و عملکرد آهسته‌تری از انواع ممیز شناور دیگر، به دست می‌دهد.

- *Extended* دقت بالاتری از انواع حقیقی دیگر عرضه می‌کند اما قابلیت انتقالش کمتر است. اگر در حال ایجاد فایل‌های داده برای اشتراک گذاری در پایگاه‌های متقاطع هستید، *Extended* را با احتیاط به کار ببرید.
- نوع *Comp* (*محاسباتی*) برای Intel CPU بومی می‌باشد و یک عدد صحیح ۶۴ بیتی را نمایندگی می‌کند. این نوع به عنوان یک عدد حقیقی طبقه بندی می‌شود، چون مانند یک نوع ترتیبی رفتار نمی‌کند. (برای مثال، نمی‌توانید یک مقدار *Comp* را نمو یا کاهش دهید.) نوع *Comp* تنها برای سازگاری با گذشته پشتیبانی می‌شود. برای کارایی و عملکرد بهتر از نوع *Int64* استفاده کنید.
- *Currency* یک نوع داده ممیز ثابت است که خطاهای گرد کردن را در محاسبات مالی کاهش می‌دهد. این نوع مانند یک عدد صحیح ۶۴ بیتی با چهار رقم کم اهمیت‌تر که به طور ضمنی مکان‌های اعشاری را نمایش می‌دهند، ذخیره می‌شود. مقادیر *Currency* زمانی که با انواع حقیقی دیگر در تخصیص‌ها و عبارات ترکیب می‌شوند، به طور خودکار به ۱۰۰۰ ضرب یا تقسیم می‌شوند.

انواع رشته (String)

یک رشته بیانگر دنباله‌ای از کاراکترهاست. پاسکال شیئی از انواع رشته از پیش تعریف شده زیر پشتیبانی می‌کند.

Table 5.5 انواع رشته

نوع	طول ماکزیمم	حافظه لازم	استفاده شده برای
ShortString	۲۵۵ کاراکتر	۲ تا ۲۵۶ بایت	سازگاری با گذشته
AnsiString	$\sim 2^{31}$ کاراکتر	۴ بایت تا ۲ گیگابایت	کاراکترهای ۸- بیت (ANSI)
WideString	$\sim 2^{30}$ کاراکتر	۴ بایت تا ۲ گیگابایت	کاراکترهای یونیکد؛ سرورهای چند کاربره و برنامه‌های کاربردی چند زبانه

AnsiString، که گاهی رشته بلند[۱] هم خوانده می‌شود، برای اغلب مقاصد ارجح تر از بقیه انواع رشته است.

انواع رشته‌ها می‌توانند در تخصیص‌ها و عبارات با هم ترکیب شوند؛ کامپایلر به طور خودکار تبدیلات لازم را انجام می‌دهد. اما رشته‌هایی که به واسطه ارجاع (by reference) به یک تابع یا روال (به عنوان پارامترهای **var** و **out**) ارسال می‌شوند، بایستی از نوع مناسب و درخور باشند. رشته‌ها می‌توانند به طور صریح به یک نوع رشته متفاوت قالب بندی شوند. (برای آگاهی از جزییات بیشتر بخش «قالب‌بندی» را در فصل ۴ ملاحظه نمایید).

واژه کلیدی **string** مانند یک شناسه نوع عام عمل می‌کند. برای مثال،

```
var S: string;
```

یک متغیر S ایجاد می‌کند که یک رشته را نگه می‌دارد. در حالت پیش فرض **{$H+}**، کامپایلر **string** را (زمانی که بدون یک عدد محصور در میان براکت‌ها بعد از آن، ظاهر شود) به صورت یک *AnsiString* تعبیر می‌کند. از فرمان **{$H–}** برای برگرداندن **string** به *ShortString* استفاده کنید.

تابع استاندارد *Length* تعداد کاراکترهای موجود در یک رشته را برمی‌گرداند. روال *SetLength* طول یک رشته را تنظیم می‌کند.

مقایسه رشته‌ها با مرتب‌سازی کاراکترها در موقعیت‌های نظیر به نظیر تعریف می‌شود. میان رشته‌هایی با طول نامساوی، هر کاراکتر در رشته بلندتر بدون یک کاراکتر نظیر به نظیر در رشته کوتاهتر یک مقدار بزرگتر از می‌گیرد. برای مثال، "AB" بزرگتر از "A" می‌باشد، بدین ترتیب، 'AB' > 'A' مقدار *True* را برمی‌گرداند. رشته‌های با طول صفر پایین‌ترین مقادیر را نگه می‌دارند.

می‌توانید یک متغیر رشته را درست همان طور که برای یک آرایه انجام می‌دهید، اندیس‌دار کنید. اگر S یک متغیر رشته باشد و i یک عبارت صحیح، S[i] بیانگر، i اُمین کاراکتر— یا اگر بخواهیم دقیق‌تر بگوییم، i اُمین بایت— واقع در S است. برای یک *ShortString* یا *AnsiString*، S[i] از نوع *AnsiChar* است؛ برای یک *WideString*، S[i] از نوع *WideChar* است. دستور MyString[2] := 'A'، مقدار A را به

[۱] Long String

دومین کاراکتر *MyString* تخصیص می‌دهد. کد زیر از تابع استاندارد *UpCase* برای تبدیل *MyString* به حروف بزرگ استفاده می‌کند.

```
var I: Integer;
begin
  I := Length(MyString);
while I > 0 do
begin
  MyString[I] := UpCase(MyString[I]);
  I := I - 1;
end;
end;
```

مواظب اندیس‌دار کردن رشته‌ها از این طریق باشید، زیرا بازنویسی انتهای یک رشته می‌تواند سبب خطاهای دسترسی شود. درضمن، از ارسال کردن اندیس‌های رشته بلند به عنوان پارامترهای **var** اجتناب کنید، زیرا منجر به کد ناکارآمدی می‌شود.

شما می‌توانید مقدار یک ثابت رشته‌ای— یا هر عبارت دیگری که یک رشته را برمی‌گرداند— را به یک متغیر تخصیص دهید. زمانی که تخصیص انجام می‌شود، طول رشته به طور پویا تغییر می‌کند. مثال‌ها:

```
MyString := 'Hello world!';
MyString := 'Hello ' + 'world';
MyString := MyString + '!';
MyString := ' '; { space }
MyString := ''; { empty string }
```

برای آگاهی از اطلاعات بیشتر، بخش‌های «رشته‌های کاراکتر» و «عملگرهای رشته» را در فصل۴ ملاحظه نمایید.

رشته‌های کوتاه

یک رشته *ShortString* صفر تا ۲۵۵ کاراکتر بلندی دارد. درحالی که طول یک رشته *ShortString* به طور پویا تغییر می‌کند، حافظه آن به طور ثابت ۲۵۶ بایت را اشغال می‌کند؛ اولین بایت طول رشته را ذخیره می‌کند و ۲۵۵ بایت باقیمانده برای کاراکترهاست. اگر S یک متغیر *ShortString* باشد، Ord(S[0]) مانند Length(S) ، طول S را برمی‌گرداند؛ تخصیص یک مقدار به S[0]، مانند فراخوانی *SetLength*، طول S را تغییر می‌دهد. *ShortString* از کاراکترهای هشت بیتی ANSI استفاده می‌کند و تنها برای سازگاری با گذشته پشتیبانی می‌شود.

پاسکال شیئی از انواع رشته کوتاه — درواقع، زیرنوع‌هایی از *ShortString* —که بیشترین طولشان در همه جا از صفر تا ۲۵۵ کاراکتر است، پشتیبانی می‌کند. اینها بوسیله اعداد داخل براکت‌ها که الحاق شده به کلمه رزرو شده **string** هستند، نمایش داده می‌شوند. برای مثال،

```
var MyString: string[100];
```

یک متغیر با نام *MyString* ایجاد می‌کند که حداکثر طولش ۱۰۰ کاراکتر است. این کد معادل با اعلان زیر است

```
type CString = string[100];
var MyString: CString;
```

متغیرهای اعلان شده به این روش تنها همان قدر حافظه ای را که نوع نیاز دارد اشغال می‌کنند— یعنی، طول ماکزیمم تعیین شده به اضافه یک بایت. در مثال ما، *MyString* از ۱۰۱ بایت، در مقایسه با ۲۵۶ بایت برای متغیری از نوع از پیش تعریف شده *ShortString* ، استفاده می‌کند.

چنان چه یک مقدار را به یک متغیر رشته کوتاه تخصیص می‌دهید، در صورتی که رشته از حداکثر طول مجاز برای نوع تجاوز کند، رشته بریده می‌شود.

توابع استاندارد *High* و *Low* روی متغیرها و شناسه‌های نوع رشته کوتاه عمل می‌کنند. تابع *High* حداکثر طول یک نوع رشته کوتاه را برمی‌گرداند، درحالی که *Low* مقدار صفر را برمی‌گرداند.

رشته‌های بلند

AnsiString، گذشته از این که یک رشته بلند خوانده می‌شود، بیانگر یک رشته با حافظه اختصاص یافته به طور پویاست که طول ماکزیمم تنها محدود به حافظه موجود و در دسترس می‌باشد. این نوع از کاراکترهای هشت بیتی ANSI استفاده می‌کند.

یک متغیر رشته بلند اشاره‌گری است که ۴ بایت از حافظه را اشغال می‌کند. زمانی که متغیر تهی می‌باشد — یعنی، زمانی که دربردارنده یک رشته با طول صفر باشد --- اشاره‌گر **nil** خواهد بود و رشته ابداً از انباره اضافی استفاده نمی‌کند. زمانی که رشته غیر تهی باشد، به بلوکی از حافظه که به طور پویا اشغال شده، اشاره می‌کند که دربردارنده مقدار رشته، یک نشانگر طول ۳۲-بیت و یک شمار

ارجاع ۳۲-بیت است. این حافظه بر روی هیپ[۱] اشغال می‌شود، اما مدیریت آن کاملاً خودکار است و به هیچ کد کاربری نیاز ندارد.

از آن جایی که متغیرهای رشته بلند اشاره‌گر هستند، دو یا چند تا از آنها می‌توانند بدون مصرف حافظه اضافی، به مقدار یکسانی ارجاع کنند. کامپایلر از این امر برای نگه داری منابع استفاده کرده و تخصیص‌ها را سریع‌تر اجرا می‌کند. هرگاه یک متغیر رشته بلند از بین می‌رود یا مقدار جدیدی به آن تخصیص داده می‌شود، شمار ارجاع رشته قدیمی (مقدار قبلی متغیر) تنزل یافته و شمار ارجاع مقدار جدید (در صورتی که یکی موجود باشد) نمو می‌یابد؛ اگر شمار ارجاع یک رشته به صفر برسد، حافظه آن آزاد می‌شود. این فرایند شمارش ارجاع[۲] خوانده می‌شود. هنگامی که از اندیس‌دار کردن برای تغییر مقدار یک کاراکتر واقع در یک رشته استفاده می‌شود، یک کپی از رشته ایجاد می‌شود اگر—اما تنها اگر— شمار ارجاع آن بزرگتر از یک باشد. این امر معناشناسی *copy-on-write* خوانده می‌شود.

رشته پهن

نوع رشته پهن (*WideString*) بیانگر رشته‌ای با حافظه اختصاص یافته به طور پویا از کاراکترهای ۱۶-بیت یونیکد است. در بیشتر مواقع این نوع مشابه با *AnsiString* است.

در Win32، نوع *WideString* با نوع *BSTR* COM سازگار است. ابزار توسعه بورلند از مشخصه‌هایی که مقادیر *AnsiString* را به *WideString* تبدیل می‌کنند، پشتیبانی می‌کند، اما ممکن است شما به طور صریح نیازمند این باشید که رشته‌های خود را به *WideString* تبدیل یا قالب‌بندی کنید.

بحثی در باره مجموعه کاراکترهای بسط یافته

ویندوز و لینوکس هردو از مجموعه کاراکترهای مولتی بایت و تک-بایت بعلاوه یونیکد پشتیبانی می‌کنند. با یک مجموعه کاراکتر تک-بایت (SBCS)[۳]، هر بایت واقع در یک رشته نمایانگر یک کاراکتر است. مجموعه کاراکتر ANSI به کار برده شده توسط خیلی از سیستم عامل‌های غربی، یک مجموعه کاراکتر تک-بایت است.

---

[۱] Heap
[۲] Reference-counting
[۳] Single-Byte Character Set

در یک مجموعه کاراکتر مولتی-بایت (MBCS)[۱]، کاراکترهای مشابه توسط یک بایت بیان شده و کاراکترهای دیگر با بیشتر از یک بایت بیان می‌شوند. اولین بایت از یک کاراکتر مولتی-بایت، ***بایت مقدم***[۲] خوانده می‌شود. به طور کلی، ۱۲۸ کاراکتر نخستین یک مجموعه کاراکتر مولتی-بایت به کاراکترهای ۷-بیت ASCII نگاشته می‌شوند. و هر بایت که مقدار ترتیبی‌اش بزرگتر از ۱۲۷ باشد، بایت مقدم یک کاراکتر مولتی-بایت است. تنها کاراکترهای تک-بایت می‌توانند دربردارنده مقدار پوچ (#0) باشند. مجموعه کاراکترهای مولتی بایت — به ویژه مجموعه کاراکترهای دابل-بایت (DBCS)[۳] — در اغلب موارد برای زبان‌های آسیایی به کار برده می‌شوند، در حالی که مجموعه کاراکتر UTF-8 استفاده شده بوسیله لینوکس یک کدگذاری مولتی-بایت یونیکد می‌باشد.

در مجموعه کاراکتر یونیکد، هر کاراکتر بوسیله دو بایت بیان می‌شود. از این رو یک رشته یونیکد دنباله‌ای از کلمه‌های دو-بایت است نه بایت‌های مجزا. در ضمن کاراکترها و رشته‌های یونیکد *کاراکترهای پهن* و *رشته‌های کاراکتر پهن* نیز خوانده می‌شوند. ۲۵۶ کاراکتر اول یونیکد به مجموعه کاراکترهای ANSI نگاشته می‌شوند. سیستم عامل ویندوز از یونیکد (UCS-2) پشتیبانی می‌کند. سیستم عامل لینوکس از UCS-4، یک ابر مجموعه UCS-2، پشتیبانی می‌کند. دلفی/کایلیکس از (UCS-2) در روی هردو پلت‌فرم، لینوکس و ویندوز، پشتیبانی می‌کند.

پاسکال شیئی از طریق انواع *Char*, *PChar*, *AnsiChar*, *PAnsiChar* و *AnsiString* از کاراکترهای تک-بایت و مولتی-بایت و رشته‌ها پشتیبانی می‌کند. اندیس‌دار کردن رشته‌های مولتی-بایت معتبر و قابل اطمینان نیست، زیرا S[i] نمایانگر i اُمین بایت (نه لزوما i اُمین کاراکتر) واقع در S می‌باشد. اگرچه توابع استاندارد اداره کننده رشته‌ها شرکای مولتی-بایت-فعالی دارند که مرتب سازی مخصوص منطقه را نیز برای کاراکترها پیاده‌سازی می‌کنند. (اسامی توابع مولتی-بایت معمولاً با *Ansi-* شروع می‌شوند. برای مثال، نسخه مولتی-بایت تابع *StrPos*، تابع *AnsiStrPos* می‌باشد.) پشتیبانی کاراکتر مولتی-بایت تابع سیستم عامل و مبتنی بر منطقه جاری می‌باشد. پاسکال شیئی کاراکترها و رشته‌های یونیکد را از طریق نوع‌های *WideChar*, *PWideChar* و *WideString* پشتیبانی می‌کند.

---

[۱] MultiByte Character Set
[۲] Lead Byte
[۳] Double-Byte Character Set

کار با رشته‌های منتهی به تهی

اغلب زبان‌های برنامه نویسی، از جمله C و C++، نیازمند نوعی داده رشته اختصاصی هستند. این زبانها و محیط‌هایی که با آن‌ها ساخته شده‌اند، به رشته‌های منتهی به تهی[1] وابسته هستند. یک رشته منتهی به تهی، یک آرایه پایه–صفر از کاراکترهاست که به NULL (#0) خاتمه می‌یابد؛ از آن جایی که آرایه نشانگر طول ندارد، اولین کاراکتر NULL انتهای رشته را مشخص می‌کند. هرگاه نیازمند این بودید که داده‌ها را با سیستم‌هایی که از آن‌ها استفاده می‌کنند به اشتراک بگذارید، برای اداره رشته‌های منتهی به تهی، می‌توانید از ساختمارهای پاسکال شیئی و روتین‌های ویژه‌ای در یونیت *SysUtils* (بخش «روتین‌های استاندارد و I/O» را در فصل ۸ ببینید) استفاده کنید. برای مثال، اعلانات نوع زیر می‌تواند برای ذخیره رشته‌های منتهی به تهی مورد استفاده قرار گیرد.

```
type
TIdentifier = array[0..15] of Char;
TFileName = array[0..259] of Char;
TMemoText = array[0..1023] of WideChar;
```

با ترکیب نوشتاری بسط یافته فعال (**{$X+}**)، می‌توانید یک ثابت رشته را به یک آرایه کاراکتر پایه–صفر با حافظه اختصاص یافته به طور استاتیک، تخصیص دهید. (آرایه‌های پویا برای این منظور مناسب نیستند.) چنان چه یک ثابت آرایه را با یک رشته که کوتاه تر از طول اعلان شده آرایه است، مقداردهی اولیه نمایید، کاراکترهای باقیمانده مقدار #0 را خواهند گرفت. برای آگاهی از اطلاعات بیشتر درباره آرایه‌ها، بخش «آرایه‌ها» را در همین فصل ملاحظه نمایید.

استفاده از اشاره‌گرها، آرایه‌ها و ثوابت رشته ای

برای دست کاری رشته‌های منتهی به تهی، اغلب لازم است که از اشاره‌گرها استفاده شود. (بخش «اشاره‌گرها و انواع اشاره‌گر» را در همین فصل ببینید.) ثوابت رشته ای با انواع *PChar* و *PWideChar* سازگار برای تخصیص هستند، که نمایانگر اشاره‌گرهایی به آرایه‌های منتهی به تهی از مقادیر *Char* و *WideChar* هستند. برای مثال،

```
var P: PChar;
...
P := 'Hello world!';
```

[1] Null-terminated strings

P به آرایه‌ای از حافظه که دربردارنده یک کپی منتهی به تهی از "Hello world!" است، اشاره می‌کند. کد قبلی معادل است با

```
const TempString: array[0..12] of Char = 'Hello world!'#0;
var P: PChar;
...
P := @TempString;
```

در ضمن می‌توانید ثوابت رشته را به هر تابع که مقدار یا پارامترهای **const** از نوع *PChar* یا *PWideChar* می‌گیرد، ارسال کنید— برای مثال StrUpper('Hello world!'). به محض تخصیصات به یک *PChar*، کامپایلر یک کپی منتهی به تهی از رشته تولید می‌کند و یک اشاره‌گر به آن کپی را به تابع می‌دهد. بالاخره، می‌توانید ثوابت *PChar* یا *PWideChar* را به تنهایی یا در یک نوع ساخت یافته، با لیترال‌های رشته مقداردهی اولیه نمایید. مثال‌ها:

```
const
Message: PChar = 'Program terminated';
Prompt: PChar = 'Enter values: ';
Digits: array[0..9] of PChar = (
'Zero', 'One', 'Two', 'Three', 'Four',
'Five', 'Six', 'Seven', 'Eight', 'Nine');
```

آرایه‌های کاراکتر پایه صفر با *PChar* و *PWideChar* سازگار هستند. هنگامی که از یک آرایه کاراکتر به جای یک مقدار اشاره‌گر استفاده کنید، کامپایلر آرایه را به یک ثابت اشاره‌گر تبدیل می‌کند که مقدارش منطبق با آدرس اولین عنصر آرایه است. برای مثال،

```
var
MyArray: array[0..32] of Char;
MyPointer: PChar;
begin
MyArray := 'Hello';
MyPointer := MyArray;
SomeProcedure(MyArray);
SomeProcedure(MyPointer);
end;
```

این کد دو مرتبه *SomeProcedure* را با مقدار یکسانی فرا می‌خواند.

یک اشاره‌گر کاراکتر می‌تواند اندیس‌دار شود طوری که انگار یک آرایه بوده است. در مثال پیشین، MyPointer[0] مقدار H را برمی‌گرداند. اندیس یک آفست اضافه شده به اشاره‌گر را قبل از این که برگشت ارجاع شود، تعیین می‌کند. (برای متغیرهای *PWideChar*، اندیس‌ها به طور خودکار به ۲ ضرب می‌شوند.) از این رو، چنان چه P یک اشاره‌گر کاراکتر باشد، P[0] معادل است با P^ و اولین کاراکتر واقع در آرایه را مشخص می‌کند، P[1] دومین کاراکتر واقع در آرایه و الی آخر؛ P[-1] بلافاصله

کاراکتر سمت چپ P[0] را مشخص می‌کند. **کامپایلر هیچ گونه وارسی دامنه را روی این اندیس‌ها انجام نمی‌دهد.**

تابع *StrUpper* استفاده از اندیس‌دار کردن اشاره‌گر را برای بازگویی سرتاسر رشته منتهی به تهی نشان می‌دهد:

```
function StrUpper(Dest, Source: PChar; MaxLen: Integer): PChar;
var
I: Integer;
begin
I := 0;
while (I < MaxLen) and (Source[I] <> #0) do
begin
Dest[I] := UpCase(Source[I]);
Inc(I);
end;
Dest[I] := #0;
Result := Dest;
end;
```

ترکیب رشته‌های پاسکال و رشته‌های منتهی به تهی

شما می‌توانید رشته‌های بلند (مقادیر *AnsiString*) و رشته‌های منتهی به تهی (مقادیر *PChar*) را در عبارات و تخصیص‌ها ترکیب کنید، و می‌توانید مقادیر *PChar* را به توابع یا روال‌هایی که پارامترهای رشته بلند می‌گیرند، ارسال کنید. دستور تخصیص S := P، جایی که S یک متغیر رشته باشد و P یک عبارت *PChar* ، یک رشته منتهی به تهی را به یک رشته بلند کپی می‌کند.

در یک عملیات باینری، چنان چه یک عملوند یک رشته بلند باشد و دیگری یک *PChar*، عملوند *PChar* به یک رشته بلند تبدیل می‌شود.

شما می‌توانید یک مقدار *PChar* را به صورت یک رشته بلند قالب بندی کنید (تبدیل صریح). هنگامی که می‌خواهید یک عملیات رشته را روی دو مقدار *PChar* انجام دهید، این عمل می‌تواند سودمند باشد. برای مثال،

```
S := string(P1) + string(P2);
```

در ضمن می‌توانید یک رشته بلند را به صورت یک رشته منتهی به تهی قالب‌بندی کنید. قواعد زیر اعمال می‌شوند.

۱ اگر S یک عبارت رشته بلند باشد، PChar(S) ، S را به صورت یک رشته منتهی به تهی برمی‌گرداند؛ PChar(S) یک اشاره‌گر به اولین کاراکتر واقع در S برمی‌گرداند.

در ویندوز: برای مثال، اگر *Str1* و *Str2* رشته‌های بلند باشند، شما می‌توانید تابع *MessageBox* را از Win32 API به صورت زیر فراخوانی کنید:

```
MessageBox(0, PChar(Str1), PChar(Str2), MB_OK);
```

(اعلان *MessageBox* در واسط یونیت *Windows* قرار دارد.)

در لینوکس: برای مثال، اگر *Str* یک رشته بلند باشد، شما می‌توانید تابع سیستم *opendir* را به صورت زیر فراخوانی کنید:

```
opendir(PChar(Str));
```

(اعلان *opendir* در واسط یونیت *Libc* قرار دارد.)

۲ در ضمن شما می‌توانید از Pointer(*S*) برای قالب‌بندی (تبدیل نوع صریح) یک رشته بلند به یک اشاره‌گر بدون نوع استفاده کنید. اما اگر *S* تهی باشد، قالب بندی **nil** را برمی‌گرداند.

۳ هنگامی که یک متغیر رشته بلند را به یک اشاره‌گر قالب‌بندی می‌کنید، اشاره‌گر تا زمانی که متغیر یک مقدار جدید بگیرد یا به خارج از دامنه برود، معتبر باقی می‌ماند. چنان چه هر عبارت رشته بلند دیگری را به یک اشاره‌گر قالب‌بندی کنید، اشاره‌گر تنها در میان دستور، جایی که تبدیل نوع انجام شده است، معتبر می‌باشد.

۴ هنگامی که یک عبارت رشته بلند را به یک اشاره‌گر قالب‌بندی می‌کنید، معمولاً اشاره‌گر باید به صورت فقط–خواندنی مورد ملاحظه قرار گیرد. تنها هنگامی که همه شرایط زیر ارضاء شوند، شما می‌توانید به درستی از اشاره‌گر برای تعدیل و اصلاح رشته بلند استفاده کنید.

۴.۱ تبدیل نوع (قالب‌بندی) عبارت، یک متغیر رشته–بلند باشد.

۴.۲ رشته تهی نباشد.

۴.۳ رشته منحصر به فرد باشد — یعنی، یک شمار ارجاع واحد داشته باشد. برای تضمین منحصر به فرد بودن رشته، روال‌های *SetLength* ، *SetString* و *UniqueString* را فراخوانی کنید.

۴.۴ رشته از وقتی که قالب‌بندی انجام شده، جرح و تعدیل نشده باشد.

۴.۵ کاراکترهای اصلاح شده همه در میان رشته باشند. مواظب باشید اندیس خارج از دامنه‌ای را روی اشاره‌گر به کار نبرید.

هنگام ترکیب کردن مقادیر *WideString* با مقادیر *PWideChar*، قواعد مشابهی قابل اعمال هستند.

انواع ساخت یافته

وهله‌های یک نوع ساخت یافته بیشتر از یک مقدار را نگه می‌دارند. نوع‌های ساخت یافته شامل مجموعه‌ها، آرایه‌ها، رکوردها و فایل‌ها و نیز نوع‌های کلاس، *class-reference* و واسط است. (برای آگاهی از اطلاعات بیشتر درباره انواع کلاس و class-reference، «کلاس‌ها و اشیاء»، را در فصل۷ ملاحظه نمایید. برای اطلاعات بیشتر درباره واسط‌ها، «واسط‌های شیء» را در فصل ۱۰ ملاحظه نمایید.) به استثنای مجموعه‌ها، که تنها مقادیر ترتیبی را نگه می‌دارند، انواع ساخت یافته می‌توانند انواع ساخت یافته دیگری را در برداشته باشند؛ یک نوع می‌تواند ترازهای نامحدودی از ترکیب‌بندی را دارا باشد.

به طور پیش‌فرض، به منظور دسترسی سریع‌تر، مقادیر یک نوع ساخت یافته بر روی مرزهای کلمه یا کلمه مضاعف تراز می‌شوند. هنگامی که یک نوع ساخت یافته را اعلان می‌کنید، برای پیاده‌سازی انباره داده فشرده، می‌توانید از کلمه رزرو شده **packed** استفاده کنید. برای مثال،

```
type TNumbers = packed array[1..100] of Real;
```

استفاده از **packed** دسترسی به داده‌ها را کندتر می‌کند و در مورد یک آرایه کاراکتر، بر سازگاری نوع تأثیر می‌گذارد. برای آگاهی از اطلاعات بیشتر فصل۱۱، «مدیریت حافظه»، را ملاحظه نمایید.

مجموعه‌ها

مجموعه، کلکسیونی از مقادیر، از نوع ترتیبی یکسانی است. مقادیر هیچ ترتیب ذاتی ندارند و ظاهر شدن یک مقدار در مجموعه، بیشتر از یک مرتبه معنادار نیست.

دامنه یک نوع مجموعه، مجموعه توانی از یک نوع ترتیبی به خصوص است، که نوع مبنا خوانده می‌شود؛ یعنی، مقادیر امکان پذیر نوع مجموعه همه زیرمجموعه ای از نوع مبنا هستند، که شامل مجموعه تهی نیز است. نوع مبنا نمی‌تواند بیشتر از ۲۵۶ مقدار ممکن داشته باشد، و رتبه آنها باید بین ۰ تا ۲۵۵ باشد. هر ساختاری به فرم

```
set of baseType
```

جایی که *baseType* یک نوع ترتیبی مناسب و درخور است، یک نوع مجموعه را مشخص می‌کند.

به علت محدودیت‌های اندازه برای انواع مبنا، انواع مجموعه معمولاً با زیردامنه‌ها تعریف می‌شوند. برای مثال، اعلان‌های

```
type
TSomeInts = 1..250;
TIntSet = set of TSomeInts;
```

یک نوع مجموعه با نام *TIntSet* ایجاد می‌کند که مقادیرش کلکسیونی از اعداد صحیح در دامنه ۱ تا ۲۵۰ می‌باشند. شما می‌توانستید همچو کاری را با کد زیر انجام دهید

```
type TIntSet = set of 1..250;
```

با این اعلان داده شده، می‌توانید مجموعه‌هایی مانند این را ایجاد کنید:

```
var Set1, Set2: TIntSet;
...
Set1 := [1, 3, 5, 7, 9];
Set2 := [2, 4, 6, 8, 10]
```

هم چنین می‌توانید مستقیماً از دستور **set of ...** در اعلان متغیرها استفاده کنید:

```
var MySet: set of 'a'..'z';
...
MySet := ['a','b','c'];
```

نمونه‌های دیگری از انواع مجموعه

```
set of Byte
set of (Club, Diamond, Heart, Spade)
set of Char;
```

عملگر **in** عضویت یک عنصر را در مجموعه، بررسی می‌کند:

```
if 'a' in MySet then ... { do something } ;
```

هر نوع مجموعه می‌تواند مجموعه تهی را نگه دارد؛ مجموعه تهی با [] نشان داده می‌شود. برای آگاهی از اطلاعات بیشتر درباره مجموعه‌ها، بخش‌های «سازنده‌های مجموعه» و «عملگرهای مجموعه» را در فصل۴ ملاحظه نمایید.

آرایه‌ها

یک آرایه نمایانگر مجموعه‌ای از عناصر اندیس‌دار از نوعی مشابه (به اسم نوع مبنا) است. از آن جایی که هر عنصر یک اندیس منحصر به فرد دارد، آرایه‌ها برخلاف مجموعه‌ها، به طور معناداری می‌توانند

مقدار مشابهی را بیشتر از یک مرتبه دارا باشند. آرایه‌ها می‌توانند حافظه را به طور پویا یا *استاتیک* به خود اختصاص دهند.

آرایه‌های استاتیک

انواع آرایه استاتیک از طریق ساختاری به شکل زیر مشخص می‌شوند

```
array[indexType1, ..., indexTypen] of baseType
```

جایی که هر *indexType* یک نوع ترتیبی است که حدودش از ۲ گیگابایت تجاوز نمی‌کند. از آن جایی که *indexTypes* آرایه را اندیس‌دار می‌کنند، تعداد عناصری که یک آرایه می‌تواند نگه دارد بوسیله حاصل ضرب اندازه‌های *indexTypes* محدود می‌شود. درعمل، معمولاً *indexTypes* زیردامنه‌هایی از اعداد صحیح هستند.

در ساده ترین حالت برای یک آرایه یک بعدی، تنها یک *indexType* وجود دارد. برای مثال،

```
var MyArray: array[1..100] of Char;
```

یک متغیر به نام *MyArray* اعلان می‌کند که آرایه‌ای با مقادیر ۱۰۰ کاراکتر را نگه می‌دارد. با اعلان داده شده، MyArray[3] سومین کاراکتر واقع در *MyArray* را نشان می‌دهد. چنان چه یک آرایه استاتیک ایجاد کنید اما هیچ مقداری به عناصر آن تخصیص ندهید، عناصر استفاده نشده هنوز هم حافظه را اشغال کرده و حاوی داده‌های تصادفی هستند؛ آنها مشابه متغیرهای مقداردهی نشده هستند.

یک آرایه چند بعدی آرایه‌ای از آرایه‌هاست. برای مثال،

```
type TMatrix = array[1..10] of array[1..50] of Real;
```

معادل است با

```
type TMatrix = array[1..10, 1..50] of Real;
```

به هر روشی که *TMatrix* اعلان شود، بیانگر یک آرایه از ۵۰۰ مقدار حقیقی است. یک متغیر *MyMatrix* از نوع *TMatrix* می‌تواند به این صورت اندیس‌دار شود: MyMatrix[2,45] ؛ یا مانند این MyMatrix[2][45]. به طور مشابه،

```
packed array[Boolean,1..10,TShoeSize] of Integer;
```

معادل است با

```
packed array[Boolean] of packed array[1..10] of packed array[TShoeSize] of Integer;
```

توابع استاندارد *Low* و *High* روی متغیرها و شناسه‌های نوع آرایه عمل می‌کنند. آنها حدود بالا و پایین نوع اولین اندیس آرایه را برمی‌گردانند. تابع استاندارد *Length* تعداد عناصر اولین بعد آرایه را برمی‌گرداند.

یک آرایه یک بعدی، بسته‌بندی شده و استاتیک از مقادیر *Char* یک رشته بسته‌بندی شده[1] خوانده می‌شود. انواع packed-string با انواع رشته سازگار هستند و با انواع دیگر packed-string هم که تعداد عناصر یکسانی داشته باشند، سازگارند. بخش «سازگاری و یگانگی هویت» را در همین فصل ملاحظه نمایید.

آرایه‌ای به فرم array[0..*x*] از *Char* یک آرایه کاراکتر پایه صفر خوانده می‌شود. آرایه‌های پایه‌صفر برای ذخیره رشته‌های منتهی به تهی به کار برده می‌شوند و با مقادیر *PChar* سازگار هستند. بخش «کار با رشته‌های منتهی به تهی» را در همین فصل ملاحظه نمایید.

آرایه‌های پویا

آرایه‌های پویا (دینامیک) طول یا اندازه ثابتی ندارند. درعوض، حافظه یک آرایه پویا، هنگامی که یک مقدار به آرایه تخصیص می‌دهید یا آن را به روال *SetLength* ارسال می‌کنید، تخصیص مجدد می‌شود. انواع آرایه پویا بوسیله ساختار زیر معرفی می‌شوند

```
array of baseType
```

برای مثال،

```
var MyFlexibleArray: array of Real;
```

آرایه پویای یک بعدی از اعداد حقیقی اعلان می‌کند. این اعلان حافظه‌ای را برای *MyFlexibleArray* تخصیص نمی‌دهد. برای ایجاد آرایه در حافظه، تابع *SetLength* را فراخوانی کنید، برای مثال، برای اعلان داده شده بالا،

---

[1] Packed string

```
SetLength(MyFlexibleArray, 20);
```

آرایه‌ای از ۲۰ عدد حقیقی را معین می‌کند، که مقادیرش از ۰ تا ۱۹ اندیس‌دار شده اند. آرایه‌های پویا همواره با اعداد صحیح اندیس‌دار می‌شوند، که این اعداد همیشه از صفر شروع می‌شوند.
متغیرهای آرایه‌های پویا به طور ضمنی اشاره‌گر هستند و با همان تکنیک شمارش-ارجاع که برای رشته‌های بلند به کار برده می‌شود، مدیریت می‌شوند. برای آزاد کردن حافظه یک آرایه پویا، **nil** را به متغیری که به آرایه اشاره می‌کند تخصیص دهید یا متغیر را به *Finalize* ارسال کنید؛ هر یک از این روش‌ها آرایه را آزاد می‌کنند، به شرط آن که ارجاعات دیگری به آن وجود نداشته باشد. آرایه‌های پویایی به طول صفر، مقدار **nil** دارند. عملگر بازگشت ارجاع (^) را به یک متغیر آرایه پویا اعمال نکنید یا آن را به روال‌های *New* یا *Dispose* ارسال ننمایید.

اگر X و Y متغیرهایی از نوعی آرایه پویای یکسان باشند، دستور X := Y ، X را به آرایه‌ای مشابهی مانند Y ارجاع می‌دهد. (در اینجا نیازی به تخصیص حافظه برای X قبل از انجام این عمل نیست.) برخلاف رشته‌ها و آرایه‌های استاتیک، آرایه‌های دینامیک قبل از اینکه نوشته شوند، به طور خودکار کپی نمی‌شوند. برای مثال، بعد از اینکه این کد اجرا شد—

```
var
A, B: array of Integer;
begin
SetLength(A, 1);
A[0] := 1;
B := A;
B[0] := 2;
end;
```

— مقدار A[0] برابر ۲ است. (اگر A و B آرایه‌های استاتیک بودند، A[0] هنوز هم ۱ می‌بود.)

تخصیص داده به یک اندیس آرایه پویا (برای مثال، MyFlexibleArray[2] := 7) آرایه را تخصیص حافظه مجدد نمی‌کند. زمان کامپایل اندیس‌های خارج از دامنه گزارش نمی‌شوند.

هنگامی که متغیرهای آرایه پویا مقایسه می‌شوند، ارجاعات آنها مقایسه می‌شوند نه مقادیر آرایه آنها. بنابراین، بعد از اجرای کد زیر

```
var
A, B: array of Integer;
begin
SetLength(A, 1);
SetLength(B, 1);
A[0] := 2;
B[0] := 2;
```

```
end;
```

A = B مقدار *False* را برمی‌گرداند در حالی که A[0] = B[0] مقدار *True* را برمی‌گرداند.

برای کوتاه کردن یک آرایه پویا، آن را به *SetLength* یا *Copy* ارسال کنید و نتیجه برگشتی را به متغیر آرایه تخصیص دهید. (معمولاً روال *SetLength* سریع‌تر عمل می‌کند.) برای مثال، اگر A یک آرایه پویا باشد، A := SetLength(A, 0, 20) همه عناصر A به جز ۲۰ عنصر اول آن را بُر می‌زند.

چنان چه یک آرایه پویا بُر زده شده باشد، می‌توانید آن را به توابع استاندارد *Length*، *High* و *Low* ارسال کنید. *Length* تعداد عناصر یک آرایه را برمی‌گرداند، *High* بالاترین اندیس آرایه را (که همان *Length*–1 است) برمی‌گرداند و *Low* صفر را برمی‌گرداند. برای حالتی که آرایه پایه–صفر است، *High* مقدار 1– را (با نتیجه متناقض که *High* < *Low*) برمی‌گرداند.

```
function CheckStrings(A: array of string): Boolean;
```

توجه در برخی اعلان‌های تابع یا روال، پارامترهای آرایه به عنوان آرایه‌ای از نوع مبنا بیان می‌شوند، بدون هیچ گونه نوع اندیس معین شده‌ای. برای مثال،

```
function CheckStrings(A: array of string): Boolean;
```

این امر نشانگر این است که تابع روی همه آرایه‌ها از نوع مبنای معین شده، عمل می‌کند، علیرغم اندازه آنها، چگونگی اندیس‌دار شدنشان یا اینکه به طور استاتیک یا پویا تخصیص حافظه شده‌اند. بخش «پارامترهای آرایه باز» را در فصل ۶ ملاحظه نمایید.

آرایه‌های دینامیک چند بعدی

برای اعلان آرایه‌های پویای چند بعدی، از ساختارهای تکراری **array of ...** استفاده کنید. برای مثال،

```
type TMessageGrid = array of array of string;
var Msgs: TMessageGrid;
```

یک آرایه دوبعدی از رشته‌ها را اعلان می‌کند. برای نمونه‌سازی این آرایه، *SetLength* را با دو آرگومان عدد صحیح فراخوانی کنید. برای مثال، چنان چه I و J متغیرهای مقدار صحیح باشند،

```
SetLength(Msgs,I,J);
```

یک آرایه I در J را معین می‌کند و برایش حافظه تخصیص می‌دهد، و Msgs[0,0] عنصری از آن آرایه را معرفی می‌کند.

شما می‌توانید آرایه‌های چند بعدی ایجاد کنید که مستطیلی نباشند. گام اول این است کـه *SetLength* را فراخوانی کرده و پارامترهایی برای n بعد اول آرایه ارسال کنید. برای مثال،

```
var Ints: array of array of Integer;
SetLength(Ints,10);
```

ده ردیف، بدون هیچ ستونی، برای *Ints* تخصیص می‌دهد. بعداً می‌توانید ستون‌ها را یکی یکی تخصیص دهید (با دادن طول متفاوتی برای آن‌ها)؛ برای مثال

```
SetLength(Ints[2], 5);
```

سومین ستون *Ints* را به اندازه پنج عدد صحیح بلندتر می‌سازد. در این نقطه (حتی اگر ستون‌های دیگـر تخصیص نشده باشند) شما می‌توانید مقادیری به سومین ستون تخصیص دهید— برای مثال،

```
Ints[2,4] := 6;
```

مثال زیر از آرایه‌های پویا (و تابع *IntToStr* که در یونیت *SysUtils* اعلان شده است) برای ایجـاد یـک ماتریس مثلثی از رشته‌ها، استفاده می‌کند.

```
var
A : array of array of string;
I, J : Integer;
begin
SetLength(A, 10);
for I := Low(A) to High(A) do
begin
SetLength(A[I], I);
for J := Low(A[I]) to High(A[I]) do
A[I,J] := IntToStr(I) + ',' + IntToStr(J) + ' ';
end;
end;
```

انواع آرایه و تخصیص‌ها

آرایه‌ها تنها اگر از نوع مشابهی باشند سازگار برای تخصیص با یک دیگر هستند. از آن جایی که پاسکال از هم ارزی نام برای نوع‌ها استفاده می‌کند، کد زیر کامپایل نخواهد شد.

```
var
Int1: array[1..10] of Integer;
Int2: array[1..10] of Integer;
...
Int1 := Int2;
```

برای این که باعث شوید، یک چنین تخصیصی به درستی کار کند، متغیرها را به صورت زیر اعلان کنید

```
var Int1, Int2: array[1..10] of Integer;
```

یا

```
type IntArray = array[1..10] of Integer;
var
Int1: IntArray;
Int2: IntArray;
```

رکوردها

یک رکورد (قابل قیاس با یک *structure* در برخی از زبان‌ها) بیانگر یک مجموعه ناهمگن از عناصر است. هر عنصر یک فیلد (*field*) خوانده می‌شود؛ اعلان یک رکورد یک اسم و نوع برای هر فیلد تعیین می‌کند. ترکیب نوشتاری/نحوی اعلان یک نوع رکورد به شکل زیر است

```
type recordTypeName = record
fieldList1: type1;
...
fieldListn: typen;
end
```

جایی که *recordTypeName* یک شناسه معتبر است، هر *type* یک نوع را معین می‌کند و هر *fieldList* یک شناسه معتبر یا لیستی از شناسه‌ها که با ویرگول از هم جدا شده اند، می‌باشد. نقطه ویرگول نهایی اختیاری است.

برای مثال، اعلان زیر یک رکورد با نام *TDateRec* ایجاد می‌کند.

```
type
TDateRec = record
Year: Integer;
Month: (Jan, Feb, Mar, Apr, May, Jun,
Jul, Aug, Sep, Oct, Nov, Dec);
Day: 1..31;
end;
```

هر *TDateRec* دارای سه فیلد است: یک مقدار صحیح با نام *Year*، مقداری از یک نوع شمارشی به نام *Month*، و دیگری عدد صحیحی بین ۱ و ۳۱ با نام *Day*. شناسه‌های *Year*، *Month* و *Day* نقش دهنده‌های فیلد مربوط به *TDateRec* هستند، و مانند متغیرها رفتار می‌کنند. اگرچه، اعلان نوع

TDateRec، هیچ حافظه‌ای را برای *Year*، *Month* و *Day* تخصیص نمی‌دهد؛ حافظه زمانی تخصیص می‌یابد که شما به صورت زیر، رکورد را نمونه‌سازی کنید:

```
var Record1, Record2: TDateRec;
```

این اعلان متغیر دو وهله از *TDateRec* به نام‌های *Record1* و *Record2* را ایجاد می‌کند.

شما می‌توانید با قیددار کردن نقش دهنده‌های فیلد با نام رکورد، به فیلدهای یک رکورد، دسترسی پیدا نمایید:

```
Record1.Year := 1904;
Record1.Month := Jun;
Record1.Day := 16;
```

یا از یک دستور **with** استفاده کنید:

```
with Record1 do
begin
Year := 1904;
Month := Jun;
Day := 16;
end;
```

اکنون می‌توانید مقادیر فیلدهای *Record1* را به *Record2* کپی کنید:

```
Record2 := Record1;
```

از آن جایی که دامنه یک نقش دهنده فیلد محدود به رکورد در جایی که رکورد رخ می‌دهد، می‌باشد، لزومی ندارد نگران تعارضات نام گذاری در بین نقش دهنده‌های فیلد و متغیرهای دیگر باشید.

به جای تعریف انواع رکورد، می‌توانید مستقیماً از ساختار **record ...** در اعلان متغیرها استفاده کنید:

```
var S: record
Name: string;
Age: Integer;
end;
```

اگرچه، اعلانی مانند این تا حد زیادی هدف اصلی رکوردها را نقض می‌کند؛ این هدف پرهیز از کدنویسی تکراری برای گروه‌های مشابهی از متغیرهاست. علاوه بر این، رکوردهایی که به طور مجزا با این شیوه اعلان می‌شوند، سازگار برای تخصیص با یک دیگر نخواهند بود حتی اگر ساختارهای آنها یکسان باشد.

بخش‌های واریانت در رکوردها

یک نوع رکورد می‌تواند یک بخش واریانت[۱] داشته باشد؛ این بخش مشابه یک دستور **case** است. بخش واریانت بایستی در پی فیلدهای دیگر واقع در اعلان رکورد بیاید.

برای اعلان یک رکورد با یک بخش واریانت، از ترکیب نوشتاری/نحوی زیر استفاده کنید.

```
type recordTypeName = record
fieldList1: type1;
...
fieldListn: typen;
case tag: ordinalType of
constantList1: (variant1);
...
constantListn: (variantn);
end;
```

بخش اول اعلان— تا بالای کلمه رزرو شده **case** — با نوع رکورد استاندارد مشابه است. بخش باقیمانده اعلان— از **case** تا نقطه ویرگول نهایی اختیاری— بخش واریانت خوانده می‌شود. در بخش واریانت،

- *tag* اختیاری است و می‌تواند هر شناسه معتبری باشد. چنان چه *tag* را از قلم انداختید، از دونقطه (**:**) بعد از آن هم صرف نظر کنید.
- *ordinalType* یک نوع ترتیبی را معرفی می‌کند.
- هر *constantList* ثابتی است که نشانگر مقداری از نوع *ordinalType* است، یا لیستی از همچو ثوابتی که با ویرگول از هم جدا شده اند. هیچ مقداری نمی‌تواند بیشتر از یک بار در *constantList* ترکیبی ظاهر شود.
- هر *variant* لیستی از اعلان‌ها برای همانند کردن *fieldList* است که با ویرگول از هم جدا شده اند: ساختارهای *type* در بخش اصلی نوع رکورد. یعنی یک *variant* شکل زیر را دارد

```
fieldList1: type1;
...
fieldListn: typen;
```

جایی که هر *fieldList* یک شناسه معتبر یا لیستی از شناسه‌هاست که با ویرگول از هم جدا شده اند، هر *type* یک نوع را معرفی می‌کند، و نقطه ویرگول نهایی اختیاری است. *types*

[۱] Variant part

نبایستی رشته‌های بلند، آرایه‌های پویا، واریانت‌ها (یعنی انواع *Variant*) یا واسط‌ها باشند، یا هیچ یک از آنها نمی‌توانند انواع ساخت یافته که محتوی رشته‌های بلند، آرایه‌های پویا، واریانت‌ها یا واسط‌ها هستند، باشند؛ اما آنها می‌توانند اشاره‌گرهایی به این انواع باشند.

رکوردها با بخش‌های واریانت از نظر ترکیب نحوی پیچیده هستند اما به طور فریبنده‌ای از نظر معناشناختی ساده هستند. بخش واریانت یک رکورد دارای *variant*های متعددی است که فضای یکسانی را در حافظه سهیم می‌شوند. شما می‌توانید هر فیلدی از هر *variant* را در هر زمانی خوانده یا در آن بنویسید؛ اما چنان چه یک فیلد واقع در یک *variant* را بنویسید و سپس به یک فیلد واقع در *variant* دیگری بنویسید، احتمال دارد داده‌های شخصی خود را رونویسی کنید. *tag*، اگر یکی موجود باشد، مانند یک فیلد اضافی (از نوع *ordinalType*) در بخش نامتغیر رکورد عمل می‌کند.

بخش‌های برای دو منظور واریانت به کار می‌روند. اول، فرض کنید شما می‌خواهید یک نوع رکورد ایجاد کنید که فیلدهایی برای انواع مختلف از داده داشته باشد، اما می‌دانید که هرگز نیازمند استفاده از همه فیلدهای واقع در یک وهله از رکورد نیستید. برای مثال،

```
type
TEmployee = record
FirstName, LastName: string[40];
BirthDate: TDate;
case Salaried: Boolean of
True: (AnnualSalary: Currency);
False: (HourlyWage: Currency);
end;
```

در اینجا منظور این است که هر کارمند یا یک حقوق دارد یا یک دستمزد ساعتی، اما نه هر دو را. پس هنگامی که شما وهله‌ای از *TEmployee* را ایجاد می‌کنید، دیگر دلیلی وجود ندارد که حافظه کافی برای هردو فیلد تخصیص دهید. در این مورد، تنها تفاوت میان *variant*ها در اسامی فیلدهاست، اما فیلدها در اصل تنها می‌توانند انواع متفاوتی را دارا باشند.

به چند مثال پیچیده‌تر توجه کنید:

```
type
TPerson = record
FirstName, LastName: string[40];
BirthDate: TDate;
case Citizen: Boolean of
True: (Birthplace: string[40]);
False: (Country: string[20];
EntryPort: string[20];
EntryDate, ExitDate: TDate);
end;
```

```
type
TShapeList = (Rectangle, Triangle, Circle, Ellipse, Other);
TFigure = record
case TShapeList of
Rectangle: (Height, Width: Real);
Triangle: (Side1, Side2, Angle: Real);
Circle: (Radius: Real);
Ellipse, Other: ();
end;
```

برای هر وهله رکورد، کامپایلر حافظه کافی را برای نگه داشتن همه فیلدهای واقع در بزرگ‌ترین *variant* تخصیص می‌دهد. *tag* اختیاری و *constantList*ها (مانند *Rectangle*، *Triangle* و ... واقع در آخرین مثال) در شیوه‌ای که کامپایلر فیلدها را مدیریت می‌کند، نقشی ندارند؛ آنها تنها برای راحتی کار برنامه‌نویس تدارک دیده شده‌اند.

دلیل دوم برای استفاده از بخش‌های واریانت این است که به شما اجازه می‌دهند با داده‌های یکسان طوری رفتار کنید که انگار به انواع متفاوتی تعلق دارند، حتی در مواردی که کامپایلر اجازه یک قالب‌بندی (تبدیل نوع صریح) را نمی‌دهد. برای مثال، چنان چه یک *Real* ۶۴-بیتی را به عنوان اولین فیلد واقع در یک *variant* و یک *Integer* ۳۲-بیتی را به عنوان اولین فیلد واقع در *variant* دیگر داشته باشید، می‌توانید یک مقدار به فیلد *Real* تخصیص داده و سپس ۳۲ بیت اول آن را به عنوان مقدار فیلد *Integer* بازخوانی کنید (مثلاً، با ارسال آن به یک تابع که به پارامترهای *Integer* نیاز دارد).

نوع‌های فایل

یک فایل مجموعه مرتبی از عناصر است که از نوع مشابهی می‌باشند. روتین‌های استاندارد I/O از نوع‌های از پیش تعریف شده *TextFile*یا *Text* استفاده می‌کنند؛ این نوع‌ها نمایانگر فایلی هستند که محتوی کاراکترهای سازماندهی شده در سطور است. برای آگاهی از اطلاعات بیشتر درباره ورودی و خروجی فایل، فصل ۸، «روتین‌های استاندارد و I/O»، را ملاحظه نمایید.

برای اعلان یک نوع فایل، از ترکیب نوشتاری/نحوی زیر استفاده کنید

```
type fileTypeName = file of type
```

جایی که *fileTypeName* هرشناسه معتبری بوده و *type* نیز یک نوع با اندازه است. انواع اشاره‌گر—خواه ضمنی یا صریح—مجاز نیستند، بنابراین یک فایل نمی‌تواند دربرگیرنده آرایه‌های پویا،

رشته‌های بلند، کلاس‌ها، اشیاء، اشاره‌گرها، واریانت‌ها، فایل‌های دیگر یا انواع ساخت‌یافته‌ای که حاوی هریک از اینها هستند، باشد. برای مثال،

```
type
PhoneEntry = record
FirstName, LastName: string[20];
PhoneNumber: string[15];
Listed: Boolean;
end;
PhoneList = file of PhoneEntry;
```

یک نوع فایل را برای گزارش و ثبت اسامی و شماره تلفن‌ها اعلان می‌کند.

هم چنین شما می‌توانید مستقیماً از ساختار **file of ...** در یک اعلان متغیر استفاده کنید. برای مثال،

```
var List1: file of PhoneEntry;
```

کلمه **file** خود نشانگر یک فایل بدون نوع است:

```
var DataFile: file;
```

برای آگاهی از اطلاعات بیشتر، بخش «فایل‌های بدون نوع» را در فصل۸ ملاحظه نمایید.

جای دادن فایل‌ها در آرایه‌ها و رکوردها مجاز نیست.

اشاره‌گرها و انواع اشاره‌گر

یک اشاره‌گر متغیری است که یک آدرس حافظه را مشخص می‌کند. زمانی که یک اشاره‌گر آدرس متغیر دیگری را نگه می‌دارد، می‌گوییم که اشاره‌گر به موقعیت آن متغیر در حافظه و یا به داده ذخیره شده در آنجا اشاره می‌کند. در مورد یک آرایه یا انواع ساخت‌یافته دیگر، یک اشاره‌گر آدرس اولین عنصر واقع در ساختار را نگه می‌دارد.

اشاره‌گرها به منظور نشان دادن نوع داده ذخیره شده در آدرس‌هایی که آن‌ها نگه می‌دارند، نوعدار می‌شوند. نوع اشاره‌گر همه منظوره یعنی *Pointer* می‌تواند بیانگر یک اشاره‌گر به هر نوع داده‌ای باشد، در حالی که انواع اشاره‌گر تخصصی‌تر تنها به انواع به خصوصی از داده‌ها اشاره (ارجاع) می‌کنند. اشاره‌گرها چهار بایت از حافظه را اشغال می‌کنند.

مقدمه ای بر اشاره‌گرها

برای دیدن چگونگی کار اشاره‌گرها، به مثال زیر نگاه کنید.

```
1  var
2  X, Y: Integer; // X and Y are Integer variables
3  P: ^Integer; // P points to an Integer
4  begin
5  X := 17; // assign a value to X
6  P := @X; // assign the address of X to P
7  Y := P^; // dereference P; assign the result to Y
8  end;
```

سطر ۲، X و Y را به عنوان متغیرهایی از نوع *Integer* اعلان می‌کند. سطر ۳، P را به عنوان یک اشاره‌گر به یک مقدار *Integer* اعلان می‌کند؛ و به معنای این است که P می‌تواند به موقعیت X یا Y اشاره کند. سطر ۵ مقداری را به X تخصیص داده و سطر ۶ آدرس X را (نشان داده شده با X@) به P تخصیص می‌دهد. بالاخره، سطر ۷ مقدار موقعیت اشاره شده بوسیله P را (نشان داده شده با P^) بازیابی کرده و آن را به Y تخصیص می‌دهد. بعد از اجرا شدن این کد، X و Y دارای مقدار مشابهی خواهند بود، یعنی ۱۷.

ضمناً عملگر @، که آن را برای گرفتن آدرس یک متغیر به کار بردیم، روی توابع و روال‌ها نیز عمل می‌کند. برای اطلاعات بیشتر، بخش‌های «عملگر @» را در فصل ۴ و «انواع رویه ای در دستورات و عبارت» را در همین فصل ببینید.

علامت ^ برای دو منظور به کار می‌رود، که هردوی آنها در مثال ما شرح داده شده اند. زمانی که قبل از یک شناسه نوع ظاهر می‌شود—

```
^typeName
```

— نوعی را نشان می‌دهد که نمایانگر اشاره‌گرهایی به متغیرهایی از نوع *typeName* است. زمانی که بعد از یک شناسه متغیر ظاهر می‌شود—

```
pointer^
```

— اشاره‌گر را برگشت ارجاع می‌زند؛ یعنی، مقدار ذخیره شده در آدرس حافظه نگه داشته شده بوسیله اشاره‌گر را برمی‌گرداند.

مثال ما ممکن است شبیه یک راه غیرمستقیم برای کپی مقدار یک متغیر به دیگری، به نظر آید— عملی که می‌توانستیم آن را با یک دستور تخصیص ساده انجام دهیم. اما اشاره‌گرها به دلایل متعددی سودمند

هستند. اولاً شناخت اشاره‌گرها به شما کمک می‌کند تا پاسکال شیئی را درک کنید، زیرا که اشاره‌گرها اغلب در پشت پرده کد عمل می‌کنند، جایی که آنها صریحاً ظاهر نمی‌شوند. هر نوع داده ای که نیازمند بلوک بزرگی از حافظه که به طور پویا تخصیص حافظه شده، باشد از اشاره‌گرها استفاده می‌کند. برای مثال، متغیرهای رشته بلند به طور ضمنی اشاره‌گر هستند، همان طور که متغیرهای کلاس این گونه‌اند. علاوه براین، برخی تکنیک‌های پیشرفته برنامه‌نویسی نیازمند استفاده از اشاره‌گرها هستند.

بالاخره، گاهی تنها راه گیر انداختن طبقه بندی نوع داده سخت گیرانه برای پاسکال شیئی اشاره‌گرها هستند. با ارجاع دادن به یک متغیر با یک اشاره‌گر (*Pointer*) همه منظوره، قالب بندی (تبدیل نوع صریح) یک *Pointer* به یک نوع اختصاصی‌تر، و برگشت ارجاع دادن آن، می‌توانید با داده‌های ذخیره شده توسط هر متغیر به گونه ای رفتار کنید که انگار به هر گونه نوعی تعلق داشته است. برای مثال، کد زیر داده ذخیره شده در یک متغیر حقیقی را به یک متغیر *integer* تخصیص می‌دهد.

```
type
PInteger = ^Integer;
var
R: Single;
I: Integer;
P: Pointer;
PI: PInteger;
begin
...
P := @R;
PI := PInteger(P);
I := PI^;
end;
```

قطعاً، اعداد صحیح و حقیقی در فرمت‌های متفاوتی ذخیره می‌شوند. این دستور تخصیص، به سادگی داده‌های باینری خام را، بدون تبدیل کردن آنها، از R به I کپی می‌کند.

علاوه بر تخصیص نتیجه یک عملیات **@**، می‌توانید از روتین‌های متعددی برای دادن یک مقدار به یک اشاره‌گر استفاده کنید. روال‌های *New* و *GetMem* یک آدرس از حافظه را به یک اشاره‌گر موجود تخصیص می‌دهند، در حالی که توابع *Addr* و *Ptr* یک اشاره‌گر به یک آدرس یا متغیر معین شده برمی‌گردانند.

اشاره‌گرهای برگشت ارجاع شده می‌توانند قیددار شده و مانند توصیف کننده‌ها عمل کنند، مانند عبارت .as in the expression P1^.Data^

کلمه رزرو شده **nil** یک ثابت ویژه است که می‌تواند به هر اشاره‌گری تخصیص داده شود. چنان چه **nil** به یک اشاره‌گر تخصیص داده شده باشد، اشاره‌گر به هیچ چیزی ارجاع (یا اشاره) نمی‌کند.

نوع‌های اشاره‌گر

با استفاده از ترکیب نوشتاری/نحوی زیر، شما می‌توانید یک اشاره‌گر را به هر نوعی اعلان کنید

```
type pointerTypeName = ^type
```

این یک تمرین پیش پا افتاده است که هنگام تعریف یک رکورد یا انواع دیگر داده، برای آن نوع یک اشاره‌گر تعریف کنید. این کار بدون کپی کردن بلوک‌های بزرگی از حافظه، دستکاری وهله‌هایی از آن نوع داده را آسان می‌کند.

برای مقاصد بسیاری، انواع اشاره‌گر استاندارد متعددی وجود دارند. *Pointer* اشاره‌گر فراگیر و چندبعدی است که می‌تواند به هر نوع داده‌ای اشاره کند. اما یک متغیر *Pointer* نمی‌تواند برگشت ارجاع داده شود؛ جای دادن علامت ^ بعد از یک متغیر *Pointer* سبب بروز یک خطای کامپایل می‌شود. برای دسترسی به داده برگشت ارجاع داده شده بوسیله یک متغیر *Pointer*، ابتدا آن را به نوع اشاره‌گر دیگری قالب‌بندی (تبدیل نوع صریح) کرده و سپس آن را برگشت ارجاع دهید.

اشاره‌گرهای کاراکتر

انواع بنیادی *PAnsiChar* و *PWideChar* بیانگر اشاره‌گرهایی به مقادیر *AnsiChar* و *WideChar* هستند. نوع کلی *PChar* بیانگر یک اشاره‌گر به *Char* (که در پیاده‌سازی فعلی‌اش، به یک *AnsiChar*) می‌باشد. اشاره‌گرهای کاراکتر برای دستکاری رشته‌های منتهی به تهی به کار برده می‌شوند. (بخش «کار با رشته‌های منتهی تهی» را در همین فصل ملاحظه نمایید.)

دیگر انواع اشاره‌گر استاندارد

یونیت‌های *System* و *SysUtils* انواع اشاره‌گر استاندارد بسیاری را اعلان می‌کنند که به طور بسیار شایعی مورد استفاده قرار می‌گیرند.

انواع اشاره‌گر منتخب اعلان شده در **System** و **SysUtils** **Table 4.7**

نوع اشاره‌گر	به متغیرهایی از نوع ..... اشاره می‌کند
PAnsiString, PString	*WideString*
PByteArray	*TByteArray* (اعلان شده در یونیت *SysUtils*). استفاده شده برای قالب‌بندی حافظه تخصیص یافته به طور پویا به منظور دسترسی به آرایه
PCurrency, PDouble, PExtended, PSingle	*Currency, Double, Extended, Single*
PInteger	*Integer*
POleVariant	*OleVariant*
PShortString	*ShortString*، هنگام انتقال کد موروثی که از نوع قدیمی PString استفاده می‌کند، سودمند است.
PTextBuf	*TTextBuf* (اعلان شده در *SysUtils*). *TTextBuf* نوع بافر داخلی واقع در یک رکورد فایل *TTextRec* است.
PVarRec	*TVarRec* (اعلان شده در یونیت *System*)
PVariant	*Variant*
PWideString	*WideString*
PWordArray	*TWordArray* (اعلان شده در یونیت *SysUtils*). برای قالب‌بندی حافظه تخصیص یافته به طور پویا برای آرایه‌هایی از مقادیر ۲-بایت به کار می‌رود.

انواع رویه‌ای

انواع رویه‌ای[۱] به شما اجازه می‌دهند تا با روال‌ها و توابع مانند مقادیری رفتار کنید که می‌توانند به متغیرها تخصیص داده شده یا به روال‌ها و توابع دیگر ارسال شوند. برای مثال، فرض کنید که یک تابع به نام *Calc* تعریف کرده‌اید که دو پارامتر *integer* را به عنوان آرگومان گرفته و یک *integer* را برمی‌گرداند :

```
function Calc(X,Y: Integer): Integer;
```

شما می‌توانید تابع *Calc* را به متغیر F تخصیص دهید:

```
var F: function(X,Y: Integer): Integer;
F := Calc;
```

اگر هدینگ هر تابع یا روالی را گرفته و شناسه بعد از کلمه **procedure** یا **function** را بردارید، آن چه که باقی می‌ماند، نام یک نوع رویه‌ای است. شما می‌توانید به طور مستقیم، یک چنین اسامی نوعی را در اعلان‌های متغیر (مانند مثال بالا) یا برای اعلان انواع جدید به کار ببرید:

```
type
TIntegerFunction = function: Integer;
TProcedure = procedure;
TStrProc = procedure(const S: string);
TMathFunc = function(X: Double): Double;
var
F: TIntegerFunction; { F is a parameterless function that returns an integer }
Proc: TProcedure;    { Proc is a parameterless procedure }
SP: TStrProc;        { SP is a procedure that takes a string parameter }
M: TMathFunc;        { M is a function that takes a Double (real) parameter and returns a Double }
procedure FuncProc(P: TIntegerFunction); { FuncProc is a procedure whose only parameter
                                           is a parameterless integer-valued function }
```

متغیرهای بالا همگی اشاره‌گرهای رویه‌ای— یعنی، اشاره‌گرهایی به آدرس یک تابع یا روال— هستند. چنان چه بخواهید به یک متد از یک وهله شیء ارجاع (یا اشاره) کنید، لازم است تا کلمات **of object** را به اسم نوع رویه‌ای اضافه کنید. برای مثال

```
type
TMethod = procedure of object;
TNotifyEvent = procedure(Sender: TObject) of object;
```

این نوع‌ها، نمایانگر اشاره‌گرهای متد هستند. یک اشاره‌گر متد در واقع یک جفت اشاره‌گر می‌باشد؛ اولی آدرس یک متد را ذخیره می‌کند و دومی یک ارجاع را به شیئی که متد به آن تعلق دارد، ذخیره می‌کند. با اعلان‌های داده شده زیر

---

[۱] Procedural type

```
type
TNotifyEvent = procedure(Sender: TObject) of object;
TMainForm = class(TForm)
procedure ButtonClick(Sender: TObject);
...
end;
var
MainForm: TMainForm;
OnClick: TNotifyEvent
```

می‌توانیم تخصیص زیر را انجام دهیم.

```
OnClick := MainForm.ButtonClick;
```

چنان چه دو نوع رویه‌ای شرایط زیر را داشته باشند، با یکدیگر سازگار هستند

- قرادادِ فراخوانی مشابه،
- مقدار برگشتی یکسان (یا این که هیچ مقدار برگشتی نداشته باشند)، و

تعداد پارامترهای یکسان، همراه با پارامترهایی که به طور یکسان در موقعیت‌های متناظر نوعدار شده‌اند. (اسامی پارامترها اهمیت ندارد.) نوع‌های اشاره‌گر رویه‌ای همواره با نوع‌های اشاره‌گر متد ناسازگار هستند. مقدار **nil** می‌تواند به هر نوع رویه‌ای تخصیص داده شود. توابع و روال‌های تودرتو (روتین‌هایی که میان روتین‌های دیگر اعلان شده اند) نمی‌توانند به عنوان مقادیر رویه‌ای به کار برده شوند، توابع و روال‌های از پیش تعریف شده نیز نمی‌توانند. چنان چه بخواهید یک روتین از پیش تعریف شده مانند *Length* را به عنوان یک مقدار رویه‌ای به کار برید، یک لفافه و پوشش برای آن بنویسید:

```
function FLength(S: string): Integer;
begin
Result := Length(S);
end;
```

نوع‌های رویه‌ای در دستورات و تخصیص‌ها

چنان چه یک متغیر رویه‌ای در سمت چپ یک دستور تخصیص قرار گرفته باشد، کامپایلر منتظر یک مقدار رویه‌ای در سمت راست خواهد بود. دستور تخصیص، متغیر سمت چپ را به یک اشاره‌گر برای تابع یا روال نشان داده شده در سمت راست تبدیل می‌کند. اگر چه، در زمینه‌های دیگر، استفاده از یک متغیر رویه‌ای، منجر به یک فراخوانی به تابع یا روال ارجاع شده (اشاره شده) می‌شود. حتی می‌توانید یک متغیر رویه‌ای را برای ارسال پارامترها به کار ببرید:

```
var
F: function(X: Integer): Integer;
I: Integer;
function SomeFunction(X: Integer): Integer;
...
F := SomeFunction; // assign SomeFunction to F
I := F(4); // call function; assign result to I
```

در دستورات تخصیص، نوع متغیر سمت چپ، تعبیر اشاره‌گرهای تابع یا روال موجود در سمت راست را مشخص می‌کند. برای مثال،

```
var
F, G: function: Integer;
I: Integer;
function SomeFunction: Integer;
...
F := SomeFunction; // assign SomeFunction to F
G := F; // copy F to G
I := G; // call function; assign result to I
```

دستور اول یک مقدار رویه ای را به F تخصیص می‌دهد. دستور تخصیص دوم آن مقدار را به متغیر دیگری کپی می‌کند. دستور سوم یک فراخوانی به تابع اشاره شده انجام می‌دهد و نتیجه را به I تخصیص می‌دهد. از آن جایی که I یک متغیر صحیح (integer) است، نه یک متغیر رویه‌ای، تخصیص نهایی در واقع تابع را فراخوانی می‌کند (که یک integer را برمی‌گرداند). در برخی از موقعیت‌ها، این که یک متغیر رویه‌ای چگونه تعبیر خواهد شد، صراحت اندکی داشته و نامشخص است. به دستور زیر توجه کنید

```
if F = MyFunction then ...;
```

در این مورد، رخداد F به یک فراخوان تابع منجر می‌شود؛ کامپایلر تابع اشاره شده توسط F را فرامی‌خواند، سپس تابع *MyFunction* را فراخوانی کرده و سپس نتایج را مقایسه می‌کند. قاعده این است که هر زمان که یک متغیر رویه‌ای در میان یک عبارت ظاهر شود، این متغیر بیانگر یک فراخوان به تابع یا روال اشاره شده است. در یک حالت جایی که F به یک روال اشاره می‌کند (که مقداری را برگشت نمی‌دهد)، یا جایی که F به یک تابع اشاره می‌کند که نیازمند یک سری پارامتر است، دستور بالا سبب یک خطای کامپایل می‌شود. برای مقایسه مقدار رویه‌ای F با *MyFunction*، از دستور زیر استفاده کنید

```
if @F = @MyFunction then ...;
```

F@، F را به متغیر اشاره‌گر بدون نوعی تبدیل می‌کند که دارای یک آدرس است و @MyFunction آدرس *MyFunction* را برمی‌گرداند. برای گرفتن آدرس حافظه یک متغیر رویه‌ای (به جای آدرس ذخیره شده در آن) از @@ استفاده کنید. برای مثال، @@F آدرس F را برمی‌گرداند. درضمن عملگر @ می‌تواند برای تخصیص یک مقدار اشاره‌گر بدون نوع به یک متغیر رویه‌ای، مورد استفاده قرار گیرد. برای مثال،

```
var StrComp: function(Str1, Str2: PChar): Integer;
...
@StrComp := GetProcAddress(KernelHandle, 'lstrcmpi');
```

تابع *GetProcAddress* را فراخوانی کرده و *StrComp* را به نتیجه ارجاع می‌دهد.

هر متغیر رویه‌ای می‌تواند مقدار **nil** را نگه دارد، که این حرف به معنای این است که این متغیر به هیچ اشاره می‌کند. اما تلاش برای فراخوانی یک متغیر رویه‌ای با مقدار **nil** یک خطا محسوب می‌شود. برای این که بررسی کنیم که آیا یک متغیر رویه‌ای تخصیص داده شده است، از تابع استاندارد *Assigned* استفاده کنید:

```
if Assigned(OnClick) then OnClick(X);
```

نوع‌های واریانت (Variant)

گاهی لازم است که داده‌هایی را دست کاری نماییم که نوع آنها در زمان کامپایل تغییر می‌کند یا نوعشان در زمان کامپایل نمی‌تواند مشخص شود. در این گونه موارد، یک راه حل این است که از متغیرها و پارامترهایی از نوع *Variant* استفاده کنیم، که بیانگر مقادیری است که می‌تواند نوعشان در زمان اجرا تغییر کند. *Variant*ها انعطاف‌پذیری بیشتری را بدست می‌دهند اما حافظه بیشتری از متغیرهای معمول مصرف می‌کنند و عملیات‌ها روی آنها کندتر از اعمال روی انواعی هستند که به طور استاتیکی مرزبندی شده‌اند. علاوه بر این، اعمال غیر مجاز روی *Variant*ها اغلب منجر به خطاهای زمان اجرا می‌شوند، در حالی که خطاهای مشابه با متغیرهای معمول در زمان کامپایل گرفته می‌شوند. ضمناً می‌توانید انواع واریانت سفارشی خود را ایجاد کنید.

به طور پیش فرض، *Variant*ها می‌توانند مقادیری از هر نوع به استثنای رکوردها، مجموعه‌ها، آرایه‌های استاتیک، فایل‌ها، کلاس‌ها، class references و اشاره‌گرها را نگه دارند. به عبارت دیگر، واریانت‌ها

می‌توانند هرچیزی به استثنای اشاره‌گرها و انواع ساخت‌یافته را نگه دارند. آنها می‌توانند واسط‌هایی را نگه دارند که متدها و خاصیت‌های آنها می‌توانند از طریق واریانت‌ها در دسترس باشند. (فصل ۱۰، «واسط‌های شیء»، را ملاحظه نمایید.) واریانت‌ها می‌توانند آرایه‌های پویا را نگه داری کنند و هم چنین می‌توانند یک نوع به خصوص از آرایه‌های استاتیک به نام یک آرایه واریانت[۱] را نگه دارند. (بخش «آرایه‌های واریانت» را در همین فصل ملاحظه نمایید.) واریانت‌ها می‌توانند با واریانت‌های دیگر و با مقادیر integer، real، string و Boolean در عبارات و تخصیص‌ها ترکیب شوند؛ کامپایلر به طور خودکار تبدیلات نوع مورد نیاز را انجام می‌دهد.

واریانت‌هایی که حاوی رشته‌ها هستند، نمی‌توانند اندیس‌دار شوند. یعنی، اگر V یک واریانت باشد که یک مقدار string را نگه داشته باشد، ترکیب V[1] سبب بروز یک خطای زمان اجرا می‌گردد.

شما می‌توانید *Variant*های شخصی خود را تعریف کنید که نوع *Variant* را برای نگه‌داری مقادیر اختیاری بسط می‌دهند. برای مثال، شما می‌توانید یک نوع رشته *Variant* تعریف کنید که اجازه اندیس‌دار شدن را دهد یا این که یک class references، نوع رکورد یا آرایه استاتیک را نگه‌داری کند. انواع *Variant* شخصی به وسیله ایجاد فرزند برای کلاس *TCustomVariantType*، تعریف می‌شوند.

یک واریانت ۱۶ بایت از حافظه را اشغال می‌کند و از یک کد نوع و یک مقدار یا اشاره‌گر به یک مقدار، از نوع تعیین شده بوسیله کد، تشکیل می‌شود. همه واریانت‌ها در زمان ایجاد با مقدار مخصوص *Unassigned* مقداردهی می‌شوند. مقدار خاص *Null* نشانگر داده ناشناخته یا مفقود است.

تابع استاندارد *VarType* کد نوع یک واریانت را برمی‌گرداند. ثابت *varTypeMask* یک ماسک بیت استفاده شده برای استخراج کد از مقدار برگشتی *VarType*، می‌باشد. به طوری که، مثلاً،

```
VarType(V) and varTypeMask = varDouble
```

چنان چه V از یک *Double* یا آرایه‌ای از *Double* تشکیل شده باشد، *True* را برمی‌گرداند. (در واقع ماسک اولین بیت را پنهان می‌کند، که نشانگر این است که آیا واریانت یک آرایه را نگه می‌دارد یا نه.) نوع رکورد تعریف شده *TVarData* واقع در یونیت *System* می‌تواند برای قالب‌بندی واریانت‌ها و دسترسی یافتن به نمایش درونی آنها مورد استفاده قرار گیرد.

---

[۱] Variant array

تبدیلات نوع Variant

همه نوع‌های integer، real، string، character و Boolean سازگار برای تخصیص با *Variant* هستند. عبارات می‌توانند صریحاً به عنوان *Variant*ها قالب‌بندی (تبدیل) شوند، و روتین‌های استاندارد *VarAsType* و *VarCast* می‌توانند برای تغییر نمایش داخلی یک واریانت به کار برده شوند. کد زیر استفاده از *Variant*ها و برخی تبدیل‌های خودکاری را که هنگام ترکیب *Variant*ها با انواع دیگر انجام می‌شوند، شرح می‌دهد.

```
var
V1, V2, V3, V4, V5: Variant;
I: Integer;
D: Double;
S: string;
begin
V1 := 1; { integer value }
V2 := 1234.5678; { real value }
V3 := 'Hello world!'; { string value }
V4 := '1000'; { string value }
V5 := V1 + V2 + V4; { real value 2235.5678}
I := V1; { I = 1 (integer value) }
D := V2; { D = 1234.5678 (real value) }
S := V3; { S = 'Hello world!' (string value) }
I := V4; { I = 1000 (integer value) }
S := V5; { S = '2235.5678' (string value) }
end;
```

کامپایلر تبدیلات نوع را بر اساس قواعد زیر انجام می‌دهد.

قواعد تبدیل نوع واریانت

Table 5.7

هدف	integer
منبع	
integer	فرمت‌های صحیح را تبدیل می‌کند
real	به نزدیک ترین عدد صحیح گرد می‌کند
string	به integer تبدیل می‌کند، اگر لازم شد همراه با بُر زدن؛ اگر رشته عددی نباشد استثناء می‌دهد
character	همانند string (بالا)
Boolean	False = 0, True = −1 (اگر Byte باشد، ۲۵۵)
Unassigned	صفر را برمی‌گرداند
Null	یک استثناء می‌دهد
هدف	

منبع	real
integer	به real تبدیل می‌کند
real	فرمت‌های real را تبدیل می‌کند
string	به real تبدیل می‌کند؛ اگر رشته عددی نباشد استثناء می‌دهد
character	همانند string (بالا)
Boolean	False = 0, True = −1
Unassigned	صفر را برمی‌گرداند
Null	استثناء می‌دهد
هدف	**string**
منبع	
integer	به نمایش رشته‌ای تبدیل می‌کند
real	با استفاده از تنظیمات منطقه‌ای به نمایش رشته‌ای تبدیل می‌کند
string	فرمت‌های رشته/کاراکتر را تبدیل می‌کند
character	همانند string (بالا)
Boolean	False = "0", True ="-1"
Unassigned	رشته تهی را برمی‌گرداند
Null	استثناء می‌دهد
هدف	**character**
منبع	
integer	همانند Integer به string
real	همانند real به string
string	همانند string به string
character	همانند string به string
Boolean	همانند Boolean به string
Unassigned	همانند Unassigned به string
Null	همانند Null به string
هدف	**Boolean**
منبع	

integer	اگر صفر باشد False و در غیر این صورت True را برمی‌گرداند
real	اگر صفر باشد False و در غیر این صورت True را برمی‌گرداند
string	اگر رشته ''false'' باشد، False برمی‌گرداند (بدون حساسیت به حالت حروف) اگر یک رشته عددی باشد که به صفر ارزیابی می‌شود False را برمی‌گرداند؛ اگر رشته ''true'' یا رشته عددی غیر صفر بود ، True را برمی‌گرداند؛ در غیر این صورت استثناء می‌دهد.
character	همانند string (بالا)
Boolean	False = False, True = True
Unassigned	False را برمی‌گرداند
Null	استثناء می‌دهد

تخصیص‌های خارج از دامنه اغلب در متغیر هدف، با گرفتن بالاترین مقدار در دامنه‌اش پیش می‌آیند. تخصیص‌ها یا قالب‌بندی‌های نامعتبر باعث بروز استثنای *EVariantError* می‌شوند.

قواعد تبدیل خاصی به نوع حقیقی *TDateTime* اعلان شده در یونیت *System* اعمال می‌شوند. زمانی که یک *TDateTime* به هر نوع دیگری تبدیل می‌شود، همانند یک *Double* نرمال رفتار می‌کند. زمانی که یک integer، real یا Boolean به یک *TDateTime* تبدیل می‌شود، ابتدا به یک *Double* تبدیل شده و سپس به صورت یک مقدار *TDateTime* خوانده می‌شود. زمانی که یک رشته به *TDateTime* تبدیل می‌شود، ابتدا با استفاده از تنظیمات منطقه‌ای به عنوان یک مقدار تاریخ–زمان تعبیر می‌شود. زمانی که یک مقدار *Unassigned* به *TDateTime* تبدیل می‌شود، با آن مانند عدد صحیح یا حقیقی برخورد می‌شود که مقدار صفر را داراست. تبدیل یک مقدار *Null* به *TDateTime* باعث بروز یک استثناء می‌شود.

در ویندوز، چنان چه یک واریانت به یک واسط COM اشاره کند، هر تلاشی برای تبدیل آن، خاصیت پیش فرض شیء را می‌خواند و آن مقدار را به نوع خواسته شده تبدیل می‌کند. اگر شیء هیچ خاصیت پیش فرضی نداشته باشد، یک استثناء بروز می‌کند.

Variantها در عبارات

همه عملگرها به استثنای ^، **is** و **in** عملوندهایی از نوع واریانت را می‌پذیرند. عملیات‌هایی که روی واریانت‌ها انجام می‌شوند مقادیر *Variant* را برمی‌گردانند؛ اگر یکی یا هردو عملوند *Null* باشند، عملیات‌ها *Null* را برمی‌گردانند، و اگر یکی یا هر دو عملوند *Unassigned* باشند، یک استثناء بروز

می‌کند. در یک عملیات باینری، اگر تنها یکی از عملوندها واریانت باشد، دیگری هم به یک واریانت تبدیل می‌شود.

نوع برگشتی یک عملیات توسط عملوندهایش مشخص می‌شود. به طور کلی، قواعدی مشابه با قواعدی که به عملوندهایی از نوع‌هایی اعمال می‌شوند که به طور استاتیک مرزبندی شده اند، به واریانت‌ها نیز اعمال می‌شوند. برای مثال، چنان چه V1 و V2 هر دو واریانت‌هایی باشند که یکی مقداری صحیح و دیگری مقداری حقیقی را نگه می‌دارد، در این صورت V1 + V2 یک واریانت را که با مقداری حقیقی مقداردهی شده، برمی‌گرداند. (بخش «عملگرها» را در فصل ۴ ببینید.) اگرچه، با *Variant*ها می‌توانید روی ترکیب‌هایی از مقادیر که اجازه ندارند از عبارات نوع دار شده به طور استاتیک استفاده کنند، عملیات‌های باینری انجام دهید. هر زمان که امکان پذیر باشد، کامپایلر واریانت‌هایی را که با هم جور نیستند، با استفاده از قواعد جمع بندی شده در جدول **5.7** تبدیل می‌کند. برای مثال، اگر V3 و V4 واریانت‌هایی باشند که یک رشته عددی و یک عدد صحیح integer را نگه می‌دارند، عبارت V3 + V4 واریانتی را که با مقداری صحیح (integer) مقدار دهی شده، برمی‌گرداند؛ قبل از این که عملیات انجام شود رشته عددی به یک integer تبدیل می‌شود.

آرایه‌های واریانت

شما نمی‌توانید یک آرایه استاتیک معمولی را به یک واریانت تخصیص دهید. در عوض، می‌توانید یک آرایه واریانت را توسط فراخوانی هر یک از توابع استاندارد *VarArrayCreate* یا *VarArrayOf* ایجاد کنید. برای مثال،

```
V: Variant;
...
V := VarArrayCreate([0,9], varInteger);
```

یک آرایه واریانت (با طول ده) از اعداد صحیح ایجاد می‌کند و آن را به متغیر V که از نوع واریانت است، تخصیص می‌دهد. آرایه می‌تواند با استفاده از V[0]، V[1] و ... اندیس‌دار شود، اما امکان ندارد که بتوان عنصری از آرایه واریانت را به عنوان یک پارامتر **var** ارسال کرد. آرایه‌های واریانت همواره با اعداد صحیح اندیس‌دار می‌شوند.

دومین پارامتر در فراخوان *VarArrayCreate* کد نوع برای نوع مبنای آرایه است. برای مشاهده لیستی از این کدها، *VarType* را در راهنمای درون خطی دلفی ملاحظه نمایید. هرگز کد *varString* را به

VarArrayCreate ارسال نکنید؛ برای ایجاد یک آرایه واریانت از رشته‌ها، از *varOleStr* استفاده کنید. واریانت‌ها می‌توانند آرایه‌های واریانت از اندازه‌ها، ابعاد و انواع مبنای متفاوت را نگه دارند. عناصر یک آرایه واریانت می‌توانند از هر نوع مجاز برای واریانت‌ها به استثنای *ShortString* و *AnsiString* باشند و چنان چه نوع مبنای آرایه *Variant* باشد، عناصر آن حتی می‌توانند ناهمگن باشند. از تابع *VarArrayRedim* برای تغییر اندازه یک آرایه واریانت استفاده کنید. روتین‌های دیگری که روی آرایه‌های واریانت عمل می‌کنند شامل *VarArrayDimCount*، *VarArrayLowBound*، *VarArrayHighBound*، *VarArrayRef*، *VarArrayLock* و *VarArrayUnlock* هستند.

چنان چه یک واریانت که حاوی یک آرایه واریانت است به واریانت دیگری تخصیص داده شود یا به عنوان یک پارامتر مقداری ارسال شود، سرتاسر آرایه کپی می‌شود. یک چنین عملیات‌هایی را به طور غیرضروری انجام ندهید، زیرا که از نظر حافظه ناکارامد هستند.

OleVariant

نوع *OleVariant* هم در پلت‌فرم لینوکس و هم پلت‌فرم ویندوز وجود دارد. تفاوت اصلی مابین *Variant* و *OleVariant* این است که *Variant* می‌تواند انواع داده ای را دربرداشته باشد که تنها برنامه جاری می‌داند که با آنها چه کار کند. *OleVariant* تنها می‌تواند حاوی انواع داده تعریف شده به طور سازگار با اتوماسیون Ole باشد که منظور انواع داده ای است که می‌توانند میان برنامه‌ها یا سرتاسر شبکه ارسال شوند بدون نگرانی درباره این که آیا انتهای دیگر (یعنی مقصد) خواهد فهمید که داده‌ها را چگونه مدیریت کند یا نه.

زمانی که شما یک *Variant* را که حاوی داده سفارشی (مانند یک رشته پاسکال یا یک نوع واریانت سفارشی جدید) است، به یک *OleVariant* تخصیص می‌دهید، کتابخانه زمان اجرا، سعی می‌کند که *Variant* را به یکی از انواع داده استاندارد *OleVariant* تبدیل کند (مانند یک رشته پاسکال که به یک رشته Ole BSTR تبدیل می‌شود). برای مثال، اگر یک واریانت حاوی یک *AnsiString* به یک *OleVariant* تخصیص داده شود، *AnsiString* یک *WideString* می‌شود. همین طور زمانی که یک *Variant* به یک پارامتر تابع *OleVariant* ارسال می‌شود، این مطلب صحیح می‌باشد.

مطابقت و یگانگی نوع

برای این که بدانیم کدام عملیات می‌تواند روی کدام عبارت انجام شود، نیازمند تشخیص انواع مختلفی از سازگاری در میان نوع‌ها و مقادیر هستیم. این امر شامل یگانگی نوع، سازگاری نوع و سازگاری برای تخصیص می‌باشد.

یگانگی نوع

مفهوم یگانگی نوع تقریباً سرراست و آسان است. زمانی که یکی از شناسه‌ها با استفاده از شناسه نوع دیگر، بدون توصیف و قید گذاری، اعلان می‌شود، آنها بیانگر نوع مشابهی هستند. از این رو، با اعلان‌های داده شده زیر

```
type
T1 = Integer;
T2 = T1;
T3 = Integer;
T4 = T2;
```

T1، T2، T3، T4 و Integer همگی نوع یکسانی را معرفی می‌کنند. برای ایجاد انواع مجزا، کلمه **type** را در اعلان تکرار کنید. برای مثال،

```
type TMyInteger = type Integer;
```

یک نوع جدید با نام *TMyInteger* ایجاد می‌کند که با *Integer* برابر نیست.

ساختارهای زبانی که همانند اسامی نوع عمل می‌کنند هر زمان که ظاهر می‌شوند یک نوع متفاوت را معرفی می‌کنند. از این رو اعلان‌های

```
type
TS1 = set of Char;
TS2 = set of Char;
```

دو نوع مجزای TS1 و TS2 را ایجاد می‌کنند. به طور مشابه، اعلان‌های متغیر

```
var
S1: string[10];
S2: string[10];
```

دو متغیر از نوع‌های مجزا ایجاد می‌کنند. برای ایجاد متغیرهایی از نوع‌های یکسان، از دستورات زیر استفاده کنید

```
var S1, S2: string[10];
```

یا

```
type MyString = string[10];
var
S1: MyString;
S2: MyString;
```

سازگاری و مطابقت نوع

هر نوع با خودش سازگار است. دو نوع مجزا در صورتی سازگار هستند که دست کم یکی از شرایط زیر را ارضاء کنند.

- هر دو نوع‌های حقیقی باشند.
- هر دو نوع‌های صحیح باشند.
- یک نوع زیردامنه‌ای از دیگری باشد.
- هردو نوع زیردامنه‌هایی از نوع مشابهی باشند.
- هر دو نوع‌های مجموعه با نوع مبنای مشابه باشند.
- هر دو نوع‌های packed-string با تعداد عناصر یکسان باشند.
- یکی نوعی رشته باشد و دیگری یک نوع رشته، packed-string یا *Char* باشد.
- یکی *Variant* باشد و دیگری یک نوع صحیح، حقیقی، کاراکتر یا بولی باشد.
- هر دو نوع‌های کلاس، class-reference یا واسط (interface) باشند، و یک نوع از دیگری مشتق شده باشد.
- هر دو نوع‌های رویه‌ای با نوع حاصل مشابه، تعداد پارامترهای مشابه و وحدت نوع میان پارامترها در موقعیت‌های متناظر باشند.
- یکی نوع *PChar* یا *PWideChar* باشد و دیگری یک آرایه کاراکتر پایه صفر به فرم array[0..*n*] از *Char* باشد.
- هر دو اشاره‌گرهایی به نوع یکسانی باشند و راهنمای کامپایلر **{$T+}** در حال اثر باشد.
- یکی نوع *Pointer* (یک اشاره‌گر بدون نوع) باشد و دیگری از هر نوع اشاره‌گری.

سازگاری برای تخصیص

سازگاری برای تخصیص یک رابطه متقارن نیست. اگر مقدار عبارت T2 در دامنه T1 قرار گیرد و دست کم یکی از شرایط زیر ارضاء شود، عبارتی از نوع T2 می‌تواند به یک متغیر از نوع T1 تخصیص داده شود.

- T1 و T2 از نوع مشابهی باشند و این نوع، نوع فایل یا نوع ساخت یافته ای که حاوی یک نوع فایل است نباشد.
- T1 و T2 انواع ترتیبی سازگاری باشند.
- T1 و T2 هردو انواع حقیقی باشند.
- T1 یک نوع حقیقی و T2 یک نوع صحیح باشد.
- T1 نوع *PChar* یا هر نوع رشته‌ای باشد و عبارت یک ثابت رشته ای باشد.
- T1 و T2 هر دو نوع‌های رشته باشند.
- T1 یک نوع رشته و T2 یک *Char* یا یک نوع packed-string باشد.
- T1 یک رشته بلند و T2 یک *PChar* باشد.
- T1 و T2 نوع‌های سازگار packed-string باشند.
- T1 و T2 نوع‌های سازگار مجموعه باشند.
- T1 و T2 نوع‌های سازگار اشاره‌گر باشند.
- T1 و T2 هر دو نوع‌های کلاس، class-reference یا واسط (interface) باشند و T2 از T1 مشتق شده باشد.
- T1 یک نوع واسط باشد و T2 یک نوع کلاس باشد که T1 را پیاده‌سازی می‌کند.
- T1 نوع *PChar* یا *PWideChar* باشد و T2 یک آرایه کاراکتر پایه صفر به فرم array[0..*n*] از *Char* باشد.
- T1 و T2 نوع‌های رویه‌ای سازگار باشند. (یک شناسه تابع یا روال — در دستورات تخصیصی خاصی— به صورت یک عبارت از یک نوع رویه‌ای رفتار می‌کند. بخش «نوع‌های رویه‌ای در عبارات و تخصیص‌ها» را در همین فصل ملاحظه نمایید.)
- T1، یک *Variant* باشد و T2 یک نوع صحیح، حقیقی، رشته، کاراکتر، بولی یا واسط (interface) باشد.
- T1 یک نوع صحیح، حقیقی، رشته، کاراکتر یا بولی باشد و T2، یک *Variant* باشد.

- T1 نوع واسط *IUnknown* یا *IDispatch* باشد و T2 یک *Variant* باشد. (اگـر T1، *IUnknown* باشد، کد نوع واریانـت بایستی *varEmpty*، *varUnknown* یا *varDispatch* باشد، و یا چنان چه T1، *varDispatch* باشد کد نوع واریانت بایستی *varEmpty* یا *varDispatch* باشد.)

اعلان نوع‌ها

یک اعلان نوع شناسه‌ای را معرفی می‌کند که نشانگر یک نوع است. ترکیب نوشتاری/نحوی برای اعلان نوع به صورت زیر است

```
type newTypeName = type
```

جایی که *newTypeName* یک شناسه معتبر است. برای مثال، با اعلان‌های داده شده

```
type TMyString = string;
```

شما می‌توانید اعلان متغیر زیر را ایجاد کنید

```
var S: TMyString;
```

دامنه یک شناسه نوع شامل خود اعلان نوع نیست (به استثنای نوع‌های اشاره‌گر). از این رو شما نمی‌توانید، برای مثال، یک نوع رکورد تعریف کنید که به طور بازگشتی از خودش استفاده کند. زمانی که یک نوع را که با یک نوع موجود یکسان است، اعلان می‌کنید، کامپایلر با شناسه جدید همانند یک اسم مستعار برای شناسه نوع قدیمی‌تر برخورد می‌کند. از این رو، با اعلان‌های داده شده زیر

```
type TValue = Real;
var
X: Real;
Y: TValue;
```

X و Y از نوع یکسان هستند؛ در زمان اجرا، هیچ راهی برای تشخیص *TValue* از *Real* وجـود نـدارد. این امر معمولاً یک پی آمد جزئی و مختصر است، اما اگر منظور شما از تعریف یک نوع جدیـد مـورد بهره برداری قرار دادن اطلاعات نوع در زمان اجرا باشد— برای مثال، متحـد کـردن یـک ویرایشـگر خاصیت با خاصیت‌های یک نوع به خصوص— تمایز قایل شدن میان «نام متفاوت» و «نوع متفاوت» مهم می‌شود. در این حالت از ترکیب نوشتاری/نحوی زیر استفاده کنید

```
type newTypeName = type type
```

برای مثال،

```
type TValue = type Real;
```

کامپایلر را وادار به ایجاد یک نوع جدید و مجزا به نام *TValue* می‌کند.

متغیرها

یک متغیر شناسه‌ای است که مقدارش می‌تواند در زمان اجرا تغییر کند. به عبارت دیگری، یک متغیر، یک نام برای یک موقعیت واقع در حافظه است؛ شما می‌توانید از نام برای خواندن و نوشتن به موقعیت حافظه استفاده کنید. متغیرها همانند لفافه‌هایی برای داده‌ها عمل می‌کنند و از آن جایی که دارای نوع هستند، به کامپایلر می‌گویند که داده‌هایی را که نگه می‌دارند، چگونه تعبیر کنند.

اعلان متغیرها

ترکیب نوشتاری/نحوی اصلی برای اعلان یک متغیر به صورت زیر است

```
var identifierList: type;
```

جایی که *identifierList* لیستی از شناسه‌های معتبر است که با ویرگول از هم جدا شده انـد و *type* هـر نوع معتبری می‌باشد. برای مثال،

```
var I: Integer;
```

متغیر I از نوع *integer* را اعلان می‌کند، در حالی که

```
var X, Y: Real;
```

دو متغیر — X و Y — از نوع *Real* را اعلان می‌کند.

اعلان‌های پشت سر هم متغیرها نیاز به تکرار واژه کلیدی **var** ندارند:

```
var
X, Y, Z: Double;
I, J, K: Integer;
Digit: 0..9;
Okay: Boolean;
```

متغیرهای اعلان شده در میان یک روال یا تابع برخی اوقات محلی[۱] خوانده می‌شوند، در حالی که به متغیرهای دیگر سراسری[۲] گفته می‌شود. متغیرهای سراسری می‌توانند در همان زمانی که اعلان می‌شوند، با استفاده از ترکیب نوشتاری/نحوی زیر، مقداردهی اولیه شوند

```
var identifier: type = constantExpression;
```

جایی که *constantExpression* هر عبارت ثابت که نمایانگر یک مقدار از نوع *type* است، می‌باشد. (برای آگاهی از اطلاعات بیشتر درباره عبارات ثابت، بخش «عبارات ثابت» را در همین فصل ملاحظه نمایید.) از این رو اعلان

```
var I: Integer = 7;
```

معادل با اعلان و دستور زیر است

```
var I: Integer;
...
I := 7;
```

اعلان‌های متغیر متعدد (مانند var X, Y, Z: Real;) نمی‌توانند حاوی مقداردهی‌های اولیه، یا اعلانهایی برای متغیرهایی از نوع فایل و واریانت باشند.

چنان چه به طور صریح یک متغیر سراسری را مقدار دهی اولیه ننمایید، کامپایلر آن را با صفر مقداردهی می‌کند. در مقابل، متغیرهای محلی، نمی‌توانند در اعلان‌هایشان مقدار دهی اولیه شوند و تا زمانی که یک مقدار به آنها تخصیص یابد حاوی مقداری تصادفی خواهند بود.

زمانی که یک متغیر را اعلان می‌کنید، شما حافظه ای را اشغال می‌کنید که چنان چه متغیر برای مدت طولانی به هیچ وجهی مورد استفاده قرار نگیرد، به طور خودکار آزاد می‌شود. به ویژه، متغیرهای محلی تنها تا زمانی که برنامه از تابع یا روال، جایی که متغیرهای محلی در آنجا اعلان شده اند، خارج شود، وجود دارند. برای آگاهی از اطلاعات بیشتر درباره متغیرها و مدیریت حافظه فصل۱۱، «مدیریت حافظه»، را ملاحظه نمایید.

آدرس‌های مطلق

---

[۱] Local

[۲] Global

شما می‌توانید متغیر جدیدی ایجاد کنید که در همان آدرسی که متغیر دیگر در آنجا مقیم است، پهلو گیرد. پس برای انجام چنین کاری، فرمان **absolute** را درست بعد از اسم نوع در اعلان متغیر جدید قرار دهید که بعد با اسم یک متغیر موجود (یعنی قبلاً اعلان شده) پی گرفته می‌شود. برای مثال،

```
var
Str: string[32];
StrLen: Byte absolute Str;
```

مشخص می‌کند که متغیر *StrLen* در همان آدرس *Str* شروع شود. از آن جایی که اولین بایت یک رشته کوتاه حاوی طول رشته است، مقدار *StrLen* برابر با طول رشته *Str* است.

شما نمی‌توانید یک متغیر را در یک اعلان **absolute** مقداردهی اولیه نمایید یا **absolute** را با هر فرمان دیگری ترکیب کنید.

متغیرهای پویا

شما می‌توانید متغیرهای دینامیک را با فراخوانی روال *GetMem* یا *New* ایجاد نمایید. یک چنین متغیرهایی در heap تخصیص داده می‌شوند و به طور خودکار مدیریت نمی‌شوند. وقتی که شما یک متغیر پویا را ایجاد می‌کنید، این وظیفه شماست که نهایتاً حافظه متغیر را آزاد کنید؛ از *FreeMem* برای تخریب متغیرهای ایجاد شده بوسیله *GetMem* استفاده کنید و از *Dispose* برای تخریب متغیرهایی که با *New* ایجاد شده‌اند استفاده کنید. روتین‌های استاندارد دیگری که روی متغیرهای دینامیک عمل می‌کنند شامل *ReallocMem*، *Initialize*، *StrAlloc* و *StrDispose* هستند.

در ضمن رشته‌های بلند، رشته‌های پهن، آرایه‌های پویا، واریانت‌ها و واسط‌ها همگی متغیرهای دینامیک تخصیص یافته در هیپ هستند، اما حافظه آنها به طور خودکار مدیریت می‌شود.

متغیرهای Thread-local

متغیرهای *Thread-local* (یا *thread*) در برنامه‌های چند ریسمانی[۱] استفاده می‌شوند. یک متغیر *Thread-local* شبیه یک متغیر سراسری است، به استثنای این که هر ریسمان از اجرا کپی مختص به خود را از

[۱] Multithreaded applications

متغیر می‌گیرد؛ این کپی نمی‌تواند از ریسمان‌های دیگر قابل دسترس باشد . متغیرهای *Thread-local* بـه جای **var** با **threadvar** اعلان می‌شوند. برای مثال،

```
threadvar X: Integer;
```

اعلان‌های متغیر ریسمان

- نمی‌توانند در میان یک روال یا تابع رخ دهند.
- نمی‌توانند حاوی مقداردهی‌های اولیه باشند.
- نمی‌توانند فرمان **absolute** را تصریح کنند.

متغیرهای ریسمان نوع اشاره‌گر یا رویه ای را ایجاد نکنید و از متغیرهـای ریسـمان در کتابخانـه‌هـای قابل بارگذاری به طور پویا — غیر از بسته‌ها (packages) — استفاده نکنید.

متغیرهای دینامیک که معمولاً توسط کامپایلر مدیریت می‌شوند — رشته‌های بلنـد، رشـته‌هـای پهـن، آرایه‌های دینامیک، واریانت‌ها و واسط‌ها — می‌توانند با **threadvar** اعلان شوند، اما کامپایلر حافظه هیپ تخصیص یافته را که توسط هر ریسمان از اجرا ایجاد شده ، به طور خودکار آزاد نمی‌کند. چنـان چه از این نوع‌های داده در متغیرهای ریسمان استفاده کنید، این وظیفه شماست که حافظـه آنهـا را آزاد کنید. برای مثال،

```
threadvar S: AnsiString;
S := 'ABCDEFGHIJKLMNOPQRSTUVWXYZ';
...
S := ''; // free the memory used by S
```

(شما می‌توانید یک واریانت را با برابر قرار دادن آن با *Unassigned* و یک واسط یا آرایه دینامیـک را با برابر قرار دادن آن با **nil**، آزاد کنید.)

ثابت‌های اعلان شده

برخی ساختارهای متفاوت زبانی منسوب به «ثابت‌ها» می‌باشند. ثابت‌های عددی (کـه *numerals* هـم خوانده می‌شوند) مانند ۱۷ و ثابت‌های رشته‌ای (که رشته‌های کاراکتر یا لیترال‌های رشـته هـم خوانـده می‌شوند) مانند 'Hello world!'؛ برای آگاهی از اطلاعات بیشتر دربـاره ثابـت‌هـای عـددی و رشـته‌ای فصل۴، «عناصر نحوی»، را ملاحظه نمایید.

ثابت‌های اعلان شده یا ثابت‌های نوع‌دار[۱] هستند یا ثابت‌های صحیح[۲]. این دو نوع از ثابت‌ها ظاهراً مشابه با یگدیگرند، اما آنها با قواعد متفاوتی کنترل می‌شوند و برای مقاصد متفاوتی به کار برده می‌شوند.

ثابت‌های صحیح

یک ثابت صحیح یک شناسه اعلان شده‌ای است که مقدارش نمی‌تواند تغییر کند. برای مثال،

```
const MaxValue = 237;
```

یک ثابت با نام *MaxValue* اعلان می‌کند که مقدار صحیح ۲۳۷ را برمی‌گرداند. ترکیب نوشتاری/نحوی برای اعلان یک ثابت صحیح به صورت زیر است

```
const identifier = constantExpression
```

جایی که *identifier* هرگونه شناسه معتبری بوده و *constantExpression* عبارتی است که کامپایلر می‌تواند آن را بدون اجرای برنامه شما ارزیابی کند. (برای آگاهی از اطلاعات بیشتر بخش «عبارات ثابت» را در همین فصل ملاحظه نمایید.)

اگر *constantExpression* یک مقدار ترتیبی را برگرداند، می‌توانید نوع ثابت اعلان شده را با استفاده از یک قالب‌بندی مقداری، مشخص کنید. برای مثال

```
const MyNumber = Int64(17);
```

ثابتی به نام *MyNumber*، از نوع *Int64*، اعلان می‌کند، که مقدار صحیح ۱۷ را برمی‌گرداند. در غیراین صورت، نوع ثابت اعلان شده، همان نوع *constantExpression* است.

- چنان چه *constantExpression* یک رشته کاراکتری باشد، ثابت اعلان شده با هر نوع رشته‌ای سازگار است. اگر رشته کاراکتری از طول یک باشد، علاوه بر این با هر نوع کاراکتری دیگر نیز سازگار است.
- چنان چه *constantExpression* یک عدد حقیقی باشد، نوع آن *Extended* است. اگر *constantExpression* یک عدد صحیح باشد، نوع آن توسط جدول زیر داده می‌شود.

---

[۱] typed constants

[۲] true constants

Table 5.8 انواع مربوط به ثابت‌های صحیح

نوع	دامنه ثابت (مبنای ۱۰)	دامنه ثابت (مبنای ۱۶)
Int64	$-2^{63}..-2147483649$	–$8000000000000000..–$80000001
Integer	-2147483648..-32769	–$80000000..–$8001
Smallint	-32768..-129	–$8000..–$81
Shortint	-128..-1	–$80..–1
0..127	0..127	0..$7F
Byte	128..255	0..$7F
0..32767	256..32767	$0100..$7FFF
Word	32768..65535	$8000..$FFFF
0..2147483647	65536..2147483647	$10000..$7FFFFFFF
Cardinal	2147483648..4294967295	$80000000..$FFFFFFFF
Int64	$4294967296..2^{63}-1$	$100000000..$7FFFFFFFFFFFFFFF

در این جا برخی نمونه‌ها از اعلان‌های ثابت آورده شده‌اند:

```
const
Min = 0;
Max = 100;
Center = (Max - Min) div 2;
Beta = Chr(225);
NumChars = Ord('Z') - Ord('A') + 1;
Message = 'Out of memory';
ErrStr = ' Error: ' + Message + '. ';
ErrPos = 80 - Length(ErrStr) div 2;
Ln10 = 2.302585092994045684;
Ln10R = 1 / Ln10;
Numeric = ['0'..'9'];
Alpha = ['A'..'Z', 'a'..'z'];
AlphaNum = Alpha + Numeric;
```

عبارات ثابت

یک عبارت ثابت، عبارتی است که کامپایلر می‌تواند آن را در جایی که رخ داده، بدون اینکه برنامه را اجرا کند، ارزیابی کند. عبارات ثابت شامل numeralها؛ رشته‌های کاراکتری؛ ثابت‌های حقیقی؛ مقادیری از انواع شمارشی؛ ثوابت ویژه *True*، *False* و **nil**؛ و عبارات ساخته شده، منحصراً از این عناصر با عملگرها، قالب‌بندی‌ها و سازنده‌های مجموعه می‌باشند. عبارات ثابت نمی‌توانند شامل متغیرها، اشاره‌گرها یا فراخوانی توابع، به استثنای فراخوانی به توابع از پیش تعریف شده زیر باشند:

Abs *High* *Low* *Pred* *Succ*

```
Chr     Length  Odd     Round   Swap
Hi      Lo      Ord     SizeOf  Trunc
```

این تعریف از عبارت ثابت در جاهای متعددی واقع در مشخصه‌های ترکیب نوشتاری/نحوی پاسکال شیئی استفاده می‌شود. عبارات ثابت برای مقداردهی متغیرهای سرتاسری، تعریف انواع زیردامنه، تخصیص رتبه به مقادیر در انواع شمارشی، تعیین مقادیر پیش فرض پارامترها، نوشتن دستورات **case** و اعلان هر دو ثابت صحیح و نوع‌دار لازم می‌شوند. نمونه‌هایی از عبارات ثابت:

```
100
'A'
256 - 1
(2.5 + 1) / (2.5 - 1)
'Borland' + ' ' + 'Developer'
Chr(32)
Ord('Z') - Ord('A') + 1
```

رشته‌های منبع

رشته‌های منبع به صورت منابع ذخیره می‌شوند و به فایل قابل اجرا یا کتابخانه متصل می‌شوند به طوری که می‌توانند بدون کامپایل مجدد برنامه اصلاح شوند.

رشته‌های منبع مانند ثوابت صحیح دیگر اعلان می‌شوند، به جز این که کلمه **resourcestring** جایگزین کلمه **const** می‌شود. عبارت سمت راست علامت = بایستی یک عبارت ثابت باشد و باید یک مقدار رشته را برگرداند. برای مثال،

```
resourcestring
CreateError = 'Cannot create file %s'; { for explanations of format specifiers, }
OpenError = 'Cannot open file %s'; { see 'Format strings' in the online Help }
LineTooLong = 'Line too long';
ProductName = 'Borland Rocks\000\000';
SomeResourceString = SomeTrueConstant;
```

کامپایلر به طور خودکار تعارضات نام‌گذاری را میان رشته‌های منبع واقع در کتابخانه‌های متفاوت رفع می‌کند.

ثابت‌های نوعدار

ثابت‌های نوعدار شده، برخلاف ثابت‌های صحیح، می‌توانند مقادیری از نوع آرایه، رکورد، رویه‌ای و اشاره‌گر باشند. ثابت‌های نوع‌دار شده نمی‌توانند در عبارت‌های ثابت ظاهر شوند.

در حالت پیش فرض کامپایلر **{$J-}**، ثابت‌های نوعدار شده نمی‌توانند مقادیر جدید تخصیص شده به آنها را دارا باشند؛ در واقع آنها، فقط–خواندنی هستند. اگرچه، در صورتی که راهنمای کامپایلر **{$J+}** در حال اثر باشد، ثابت‌های نوعدار شده می‌توانند مقادیر جدید تخصیص شده به آنها را دارا باشند؛ آنها در اصل مانند متغیرهای مقداردهی شده رفتار می‌کنند. یک ثابت نوعدار را به این صورت اعلان کنید:

```
const identifier: type = value
```

جایی که *identifier* هر گونه شناسه معتبری بوده و *type* هر نوعی به استثنای فایل‌ها و واریانت‌ها می‌باشد و *value* یک عبارت از نوع *type* است. برای مثال،

```
const Max: Integer = 100;
```

در اغلب موارد، *value* بایستی یک عبارت ثابت باشد؛ اما اگر *type* یک نوع آرایه، رکورد، رویه‌ای یا اشاره‌گر باشد، قواعد به خصوصی اعمال می‌شود.

ثابت‌های آرایه

برای اعلان یک ثابت آرایه‌ای، مقادیر عناصر آرایه را که با ویرگول از هم جدا می‌شوند، در میان پارانتزهایی در انتهای اعلان محصور کنید. این مقادیر بایستی توسط عبارات ثابت بیان شده باشند. برای مثال،

```
const Digits: array[0..9] of Char = ('0', '1', '2', '3', '4', '5', '6', '7', '8', '9');
```

یک ثابت نوع‌دار با نام *Digits* اعلان می‌کند که آرایه‌ای از کاراکترها را نگه می‌دارد.

اغلب آرایه‌های کاراکتر پایه–صفر نمایانگر رشته‌های منتهی به تهی[۱] هستند و به این خاطر ثابت‌های رشته می‌توانند برای مقداردهی آرایه‌های کاراکتر مورد استفاده قرار گیرند. از این رو اعلان بالا به راحتی می‌تواند به صورت زیر بیان شود

```
const Digits: array[0..9] of Char = '0123456789';
```

[۱] Null-terminated strings

برای تعریف یک ثابت آرایه چند بعدی، مقادیر هر بعد را در مجموعه مجزایی از پارانتزها، که به وسیله ویرگول از هم جدا شده‌اند، محصور کنید. برای مثال،

```
type TCube = array[0..1, 0..1, 0..1] of Integer;
const Maze: TCube = (((0, 1), (2, 3)), ((4, 5), (6,7)));
```

یک آرایه با نام *Maze* ایجاد می‌کند، جایی که

```
Maze[0,0,0] = 0
Maze[0,0,1] = 1
Maze[0,1,0] = 2
Maze[0,1,1] = 3
Maze[1,0,0] = 4
Maze[1,0,1] = 5
Maze[1,1,0] = 6
Maze[1,1,1] = 7
```

ثابت‌های آرایه نمی‌توانند دربردارنده مقادیر نوع فایل در هر ترازی باشند.

ثابت‌های رکورد

برای اعلان یک ثابت رکورد، مقدار هر فیلد را— مانند *fieldName*: *value*، با تخصیصات فیلد جدا شده توسط نقطه ویرگول‌ها— در میان پارانتزهایی در انتهای اعلان، مشخص کنید. مقادیر بایستی توسط عبارت‌های ثابت بیان شوند. فیلدها باید به ترتیبی که در اعلان نوع رکورد ظاهر می‌شوند، لیست شوند و فیلد tag، چنان چه یک چنین فیلدی در آنجا موجود باشد، باید یک مقدار تعیین شده داشته باشد؛ اگر رکورد یک بخش واریانت داشته باشد، تنها به واریانت انتخاب شده توسط فیلد tag می توان مقادیری را تخصیص داد. مثال‌ها:

```
type
TPoint = record
X, Y: Single;
end;
TVector = array[0..1] of TPoint;
TMonth = (Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec);
TDate = record
D: 1..31;
M: TMonth;
Y: 1900..1999;
end;
const
Origin: TPoint = (X: 0.0; Y: 0.0);
Line: TVector = ((X: -3.1; Y: 1.5), (X: 5.8; Y: 3.0));
SomeDay: TDate = (D: 2; M: Dec; Y: 1960);
```

ثابت‌های رکورد نمی‌توانند دربردارنده مقادیر نوع فایل در هر ترازی باشند.

ثابت‌های رویه‌ای

برای اعلان یک ثابت رویه‌ای، اسم یک تابع یا روال را که با نوع اعلان شده ثابت سازگار باشد، تعیین کنید. برای مثال،

```
function Calc(X, Y: Integer): Integer;
begin
...
end;
type TFunction = function(X, Y: Integer): Integer;
const MyFunction: TFunction = Calc;
```

با این اعلان‌های داده شده، شما می‌توانید از ثابت رویه‌ای *MyFunction* در یک فراخوان تابع استفاده کنید:

```
I := MyFunction(5, 7)
```

در ضمن می‌توانید مقدار **nil** را به یک ثابت رویه‌ای تخصیص دهید.

ثابت‌های اشاره‌گر

زمانی که یک ثابت اشاره‌گر را اعلان می‌کنید، بایستی آن را با مقداری که می‌تواند در زمان کامپایل تجزیه شود — دست کم به عنوان یک آدرس نسبی — مقداردهی اولیه نمایید. در اینجا سه راه برای انجام این کار وجود دارد: با عملگر **@**، با **nil**، و (اگر ثابت از نوع *PChar* باشد) با یک لیترال رشته. برای مثال، اگر I یک متغیر سراسری از نوع *Integer* باشد، شما می‌توانید یک ثابت مانند زیر اعلان کنید

```
const PI: ^Integer = @I;
```

کامپایلر می‌تواند آن را تجزیه کند زیرا متغیرهای سراسری بخشی از قطعه کد هستند. از این رو توابع و ثوابت سرتاسری هستند:

```
const PF: Pointer = @MyFunction;
```

از آن جایی که لیترال‌های رشته، به صورت ثوابت سراسری تخصیص داده شده‌اند، شما می‌توانید یک ثابت *PChar* را با یک لیترال رشته مقداردهی کنید :

```
const WarningStr: PChar = 'Warning!';
```

آدرس‌های متغیرهای محلی (stack-allocated) و دینامیک (heap-allocated) نمی‌توانند به ثابت‌های اشاره‌گر تخصیص داده شوند.

فصل ۶

توابع و روال‌ها

روال‌ها و توابع، که همگی منتسب به روتین‌ها هستند، بلـوک‌های دستوری توداری هستند که می‌تواننـد از موقعیت‌های مختلف یک برنامه فراخوانده شوند. یک تابع، روتینی است که هر زمان اجرا شود یـک مقدار برمی‌گرداند. یک روال روتینی است که مقدار برگشتی ندارد.

فراخوان‌های توابع، از آن جایی که یـک مقـدار را برمـی‌گردانـند، مـی‌تواننـد بـه صـورت عبـارت در عملیات‌ها و تخصیص‌ها مورد استفاده قرار گیرند. برای مثال،

```
I := SomeFunction(X);
```

تابع *SomeFunction* را فراخوانده و نتیجه را به I تخصیص می‌دهد. فراخوان‌های توابـع نمـی‌تواننـد در سمت چپ یک عبارت تخصیص ظاهر شوند.

فراخون‌های روال‌ها — و زمانی که ترکیب بسط یافته فعال باشد (**{$X+}**)، فراخوان‌هـای توابـع — خود می‌توانند به عنوان دستورات کامل به کار برده شوند. برای مثال،

```
DoSomething;
```

روتین *DoSomething* را فرامی‌خوانـد؛ اگـر *DoSomething* یـک تـابع باشـد، مقـدار برگشـتی آن دور انداخته می‌شود. روال‌ها و توابع می‌توانند خودشان را به طور بازگشتی فراخوانی کنند.

اعلان روال‌ها و توابع

زمانی که یک تابع یا روال را اعلان می‌کنید، شما نام آن روتین، تعداد و نوع پارامترهایی که می‌گیرد، و در مورد یک تابع، نوع مقدار برگشتی‌اش را تعیین می‌کنید؛ این بخـش از اعـلان برخـی اوقـات نمونـه اولیه[۱]، هدر یا هدینگ خوانده می‌شود. بعد از آن بلوکی از کد را که هنگام فراخوانی تابع یا روال انجـام می‌شود، می‌نویسید؛ این بخش برخی اوقات بلوک یا بدنه روتین خوانده می‌شود.

روال استاندارد *Exit* می‌تواند در میان بدنه هر روال یا تابعی ظاهر شود. *Exit* اجـرای روتـین را از هـر جایی که ظاهر می‌شود، متوقف کرده و بلافاصله کنترل برنامه را به نقطه‌ای که روتین فراخوانی شده بود منتقل می‌کند.

اعلان‌های روال

اعلان یک روال قالب زیر را دارد

```
procedure procedureName(parameterList); directives;
localDeclarations;
begin
statements
end;
```

جایی که *procedureName* هر شناسه معتبری بوده و *statements* دنباله‌ای از دستورات است که هنگام فراخوانی روال اجرا می‌شوند، و (*parameterList*)، *directives* و *localDeclarations* اختیاری هستند.

- برای اطلاعات بیشتر درباره *parameterList*، بخش «پارامترها» را در همین فصل ببینید.
- برای اطلاعات بیشتر درباره *directives*، بخش‌های «اعلان‌های خارجی»، «اعلان‌های واسط و Forward»، «قراردادهای فراخوانی»، «سربارگذاری روال‌ها و توابع» و «نوشتن کتابخانه‌هـای قابل بارگذاری به طور پویا» را در همین فصل ببینید. چنان چه بیشتر از یک دستور را ضمیمه کردید، آنها را با نقطه ویرگول از هم جدا کنید.

---

[۱] Prototype

- برای اطلاعات بیشتر درباره *localDeclarations*، که شناسه‌های محلی را اعلان می‌کند، بخش «اعلانهای محلی» را در همین فصل ملاحظه نمایید.

در اینجا یک نمونه از اعلان روال آورده شده است:

```
procedure NumString(N: Integer; var S: string);
var
V: Integer;
begin
V := Abs(N);
S := '';
repeat
S := Chr(V mod 10 + Ord('0')) + S;
V := V div 10;
until V = 0;
if N < 0 then S := '-' + S;
end;
```

با این اعلان داده شده، شما می‌توانید روال *NumString* را به این صورت فراخوانی کنید:

```
NumString(17, MyString);
```

این فراخوان روال مقدار "17" را به *MyString* (که باید یک متغیر **string** باشد) تخصیص می‌دهد.

در میان بلوک دستور یک روال، می‌توانید از دیگر متغیرها و شناسه‌های اعلان شده در بخش *localDeclarations* روال، استفاده کنید. در ضمن شما می‌توانید از اسامی پارامتر لیست پارامتر (مانند N و S در مثال پیشین) استفاده کنید؛ لیست پارامتر مجموعه‌ای از متغیرهای محلی را تعریف می‌کند، پس سعی نکنید که اسامی پارامترها را در بخش *localDeclarations* اعلان مجدد کنید. بالاخره، می‌توانید هر شناسه‌ای را که دامنه‌اش در اعلان روال می‌افتد، به کار ببرید.

اعلان‌های تابع

اعلان یک تابع مشابه اعلان یک روال است به جز این که یک نوع برگشتی و یک مقدار برگشتی را تعیین می‌کند. اعلانات تابع قالب زیر را دارند

```
function functionName(parameterList): returnType; directives;
localDeclarations;
begin
statements
end;
```

جایی که *functionName* هر شناسه معتبری بوده و *returnType* هر گونه نوعی می‌باشد و *statements* دنباله‌ای از دستورات است که هنگام فراخوانی تابع اجرا می‌شوند و (*parameterList*)، *directives* و *localDeclarations* اختیاری هستند.

- برای اطلاعات بیشتر درباره *parameterList*، بخش «پارامترها» را در همین فصل ملاحظه نمایید.
- برای اطلاعات بیشتر درباره *directives*، بخش‌های «اعلان‌های خارجی»، «اعلان‌های واسط و Forward»، «قراردادهای فراخوانی»، «سربارگذاری روال‌ها و توابع» و «نوشتن کتابخانه‌های قابل بارگذاری به طور پویا» را در همین فصل ملاحظه نمایید. چنان چه شما بیشتر از یک دستور را ضمیمه کردید، آنها را با نقطه ویرگول از هم جدا کنید.
- برای اطلاعات بیشتر درباره *localDeclarations*، که شناسه‌های محلی را اعلان می‌کند، بخش «اعلانهای محلی» را در همین فصل ملاحظه نمایید.

بلوک دستورات تابع با قواعد مشابهی که به روال‌ها اعمال می‌شوند، کنترل می‌گردند. در میان بلوک دستور، شما می‌توانید از دیگر متغیرها و شناسه‌های اعلان شده در بخش *localDeclarations* تابع، استفاده کنید. در ضمن می‌توانید از اسامی پارامتر لیست پارامتر استفاده کنید؛ بالاخره، می‌توانید هر شناسه‌ای را که دامنه‌اش در اعلان تابع می‌افتد، استفاده کنید. علاوه بر این، اسم تابع خودش همانند متغیری خاص که مقدار برگشتی تابع را نگه می‌دارد، عمل می‌کند، همان طور که متغیر از پیش تعریف شده *Result* این کار را انجام می‌دهد. برای مثال،

```
function WF: Integer;
begin
WF := 17;
end;
```

یک تابع ثابت با نام WF تعریف می‌کند که هیچ پارامتری نمی‌گیرد و همواره یک مقدار صحیح ۱۷ را برمی‌گرداند. این اعلان معادل است با

```
function WF: Integer;
begin
Result := 17;
end;
```

در این جا اعلان یک تابع با پیچیدگی بیشتری آورده شده است:

```
function Max(A: array of Real; N: Integer): Real;
var
X: Real;
```

```
I: Integer;
begin
X := A[0];
for I := 1 to N - 1 do
if X < A[I] then X := A[I];
Max := X;
end;
```

شما می‌توانید مکرراً یک مقدار به *Result* یا به نام تابع در میان یک بلوک دستور تخصیص دهید، به شرطی که تنها مقادیری را که با نوع برگشتی اعلان شده مطابقت دارند، تخصیص دهید. زمانی که اجرای تابع خاتمه می‌یابد، هر مقداری که دیرتر به *Result* یا به نام تابع تخصیص یافته است، مقدار برگشتی تابع می‌شود. برای مثال،

```
function Power(X: Real; Y: Integer): Real;
var
I: Integer;
begin
Result := 1.0;
I := Y;
while I > 0 do
begin
if Odd(I) then Result := Result * X;
I := I div 2;
X := Sqr(X);
end;
end;
```

Result و نام تابع همواره بیانگر مقدار یکسانی هستند. از این رو

```
function MyFunction: Integer;
begin
MyFunction := 5;
Result := Result * 2;
MyFunction := Result + 1;
end;
```

مقدار ۱۱ را برمی‌گرداند. اما *Result* به طور کامل قابل تعویض با نام تابع نیست. زمانی که نام تابع در سمت چپ یک دستور تخصیص ظاهر می‌شود، کامپایلر فرض می‌کند که این شناسه (مانند *Result*) برای ردیابی مقدار برگشتی به کار رفته است؛ چنان چه نام تابع هرجای دیگری در بلوک دستور ظاهر شود، کامپایلر آن را به عنوان یک فراخوان بازگشتی به خود تعبیر می‌کند. از سوی دیگر، *Result* می‌تواند به عنوان یک متغیر در عملیات‌ها، قالب‌بندی‌ها، سازنده‌های مجموعه، اندیس‌ها و فراخوان به روتین‌های دیگر به کار رود.

مادامی که ترکیب نوشتاری/نحوی بسط یافته (**{$X+}**) فعال باشد، *Result* به طور ضمنی در هر تابعی اعلان می‌شود. سعی نکنید تا آن را اعلان مجدد نمایید. اگر اجرا خاتمه یابد بدون این که تخصیصی به *Result* یا نام تابع صورت گیرد، در این صورت مقدار برگشتی تابع تعریف نشده خواهد بود.

قراردادهای فراخوانی

زمانی که یک روال یا تابع را اعلان می‌کنید، شما می‌توانید یک قرارداد فراخوانی را با استفاده از یکی از راهنماهای **register**، **pascal**، **cdecl**، **stdcall** و **safecall** تصریح کنید. برای مثال،

```
function MyFunction(X, Y: Real): Real; cdecl;
...
```

قراردادهای فراخوانی ترتیبی را که در آن، پارامترها به روتین ارسال می‌شوند مشخص می‌کنند. در ضمن آنها بر انتقال پارامترها از پشته[۱]، استفاده از ثبات‌ها برای ارسال پارامترها و مدیریت استثناها و خطاها اثر می‌گذارند. قرارداد فراخوانی پیش فرض، **register** است.

- قراردادهای **register** و **pascal** پارامترها را از چپ به راست ارسال می‌کنند؛ یعنی نخست سمت چپ‌ترین پارامتر ارزیابی و ارسال می‌شود و در انتها سمت راست‌ترین پارامتر ارزیابی و ارسال می‌شود. قراردادهای **cdecl**، **stdcall** و **safecall** پارامترها را از راست به چپ ارسال می‌کنند.
- برای همه قراردادها به استثنای **cdecl**، روال یا تابع به مجرد برگشت، پارامترها را از پشته حذف می‌کند. با قرارداد **cdecl**، فراخواننده زمانی که فراخوان برگردانده شد، پارامترها را از پشته حذف می‌کند.
- قرارداد **register** تا سه ثبات از CPU را برای ارسال پارامترها مصرف می‌کند، در حالی که قراردادهای دیگر همه پارامترها را روی پشته ارسال می‌کنند.
- قرارداد **safecall** اخطار "firewalls" را پیاده‌سازی می‌کند. در ویندوز این امر اعلام خطای COM فرایند درونی را پیاده‌سازی می‌کند.

---

[۱] Stack

جدول زیر قراردادهای فراخوانی را جمع‌بندی می‌کند.

قراردادهای فراخوانی **Table 6.1**

دستور	ترتیب پارامتر	پاک سازی	پارامترها در رجیسترها را ارسال می‌کند؟
register	چپ به راست	روتین	بله
pascal	چپ به راست	روتین	خیر
cdecl	راست به چپ	فراخواننده	خیر
stdcall	راست به چپ	روتین	خیر
safecall	راست به چپ	روتین	خیر

قرارداد پیش فرض **register** کارامدترین است، زیرا معمولاً از ایجاد یک چارچوب پشته پرهیز می‌کند. (متدهای دسترسی برای خاصیت‌های منتشرشده[1] بایستی از **register** استفاده کنند.) زمانی که توابعی را از کتابخانه‌های اشتراکی نوشته شده در C یا C++ فرامی‌خوانید، قرارداد **cdecl** سودمند خواهد بود، در حالی که **stdcall** و **safecall** به طور کلی، برای فراخوان به کد بیرونی توصیه می‌شوند. در ویندوز، APIهای سیستم عامل **stdcall** و **safecall** هستند. در سیستم عامل‌های دیگر به طور کلی از **cdecl** استفاده می‌کنند. (توجه کنید که **stdcall** کارآمدتر از **cdecl** می‌باشد.)

برای اعلان کردن متدهای واسط–دوگانه (فصل۱۰، «واسط‌های شیء»، را ملاحظه نمایید) باید قرارداد **safecall** به کار برده شود. قرارداد **pascal** برای سازگاری با گذشته پشتیبانی می‌شود. برای اطلاعات بیشتر درباره قراردادهای فراخوانی، فصل۱۲، «کنترل برنامه»، را ملاحظه نمایید.

راهنماهای **near**، **far** و **export** اشاره به قراردادهای فراخوانی در برنامه نویسی ویندوز ۱۶بیتی دارند. این راهنماها در برنامه‌های ۳۲بیتی اثری ندارند و تنها برای سازگاری با گذشته پشتیبانی می‌شوند.

[1] Published

اعلان‌های واسط و Forward

دستور **forward** بلوکی، شامل اعلان متغیرهای محلی و دستورها را در اعلان یک روال یا تابع جایگزین می‌کند. برای مثال،

```
function Calculate(X, Y: Integer): Real; forward;
```

یک تابع به نام *Calculate* اعلان می‌کند. یک جایی بعد از اعلان **forward**، روتین باید در یک تعریف اعلان[۱] که شامل یک بلوک است، اعلان مجدد شود. تعریف اعلان مربوط به *Calculate* احتمالاً شبیه این خواهد بود:

```
function Calculate;
...    { declarations }
begin
...    { statement block }
end;
```

معمولاً یک تعریف اعلان لازم نیست که لیست پارامترها یا نوع برگشتی روتین را تکرار کند، اما چنان چه آنها را تکرار کند، آنها بایستی با آن چه در اعلان **forward** است دقیقاً مطابقت داشته باشند (به جز این که می‌توان پارامترهای پیش فرض را از قلم انداخت). اگر اعلان **forward** یک تابع یا روال سربارگذاری شده را تعیین کند، در این صورت تعریف اعلان بایستی لیست پارامترها را تکرار کند. میان یک اعلان **forward** و تعریف اعلانش، می‌توانید هیچ چیزی به جز اعلان‌های دیگر جای ندهید. تعریف اعلان می‌تواند یک اعلان **external** یا **assembler** باشد، اما نمی‌تواند اعلان **forward** دیگری باشد.

هدف از یک اعلان **forward** بسط دادن دامنه شناسه یک روال یا تابع به یک نقطه قبل‌تر در کد منبع می‌باشد. این کار به روال‌ها و توابع دیگر اجازه می‌دهد که روتینی را که اعلان **forward** شده است قبل از این که واقعاً تعریف شده باشد، فراخوانی کنند. گذشته از این که اعلان‌های **forward** به شما اجازه می‌دهند تا کد خود را انعطاف‌پذیرتر کنید، برخی اوقات برای بازگشت‌های متقابل لازم می‌شوند.

استفاده از راهنمای **forward** در بخش **interface** یک یونیت، قانونی نیست. گرچه، هدرهای تابع و روال در بخش **interface**، مانند اعلان‌های **forward** رفتار می‌کنند و باید تعریف اعلان‌هایی در

---

[۱] Defining declaration

بخش پیاده‌سازی (**implementation**) داشته باشند. یک روتین اعلان شده در بخش واسط (**interface**) از هرجای دیگر یونیت و از هر یونیت یا برنامه دیگری که از یونیت، جایی که اعلان شده است، استفاده می‌کند قابل دسترسی است.

اعلان‌های بیرونی

راهنمای **external**، که بلوکی را در اعلان یک روال یا تابع جایگزین می‌کند، به شما اجازه می‌دهد تا روتین‌هایی را که به طور جداگانه، بیرون از برنامه شما کامپایل شده‌اند، فراخوانی کنید. روتین‌های بیرونی می‌توانند از فایل‌های شیئی[۱] یا کتابخانه‌های قابل بارگذاری به طور پویا[۲]، بیایند.
هنگامی‌که یک تابع C++ را وارد می‌کنید که تعداد متغیری پارامتر می‌پذیرد، از راهنمای **varargs** استفاده کنید. برای مثال،

```
function printf(Format: PChar): Integer; cdecl; varargs;
```

راهنمای **varargs** تنها با روتین‌های بیرونی و صرفاً با قرارداد فراخوانی **cdecl** کار می‌کند.

اتصال به فایل‌های شیئی

برای فراخوانی روتین‌ها از یک فایل شیئی که به طور مجزا کامپایل شده، ابتدا فایل شیئ را با استفاده از فرمان کامپایلر **$L** (یا **$LINK**) به برنامه خود متصل نمایید. برای مثال،

```
On Windows:    {$L BLOCK.OBJ}
On Linux:      {$L block.o}
```

BLOCK.OBJ را (در ویندوز) یا block.o را (در لینوکس) به برنامه یا یونیت، جایی که رخ می‌دهد، پیوند می‌دهد. بعد، توابع و روال‌هایی را که می‌خواهید فراخوانی کنید، اعلان کنید:

```
procedure MoveWord(var Source, Dest; Count: Integer); external;
procedure FillWord(var Dest; Data: Integer; Count: Integer); external;
```

حال می‌توانید روتین‌های *MoveWord* و *FillWord* را از BLOCK.OBJ (ویندوز) یا block.o (لینوکس) فراخوانی کنید.

---

[۱] Object files

[۲] Dynamically Loadable Libraries (DLLs)

خیلی اوقات اعلان‌هایی مانند یکی از اعلان‌های بالایی برای دسترسی به روتین‌های بیرونی نوشته شده در زبان اسمبلی به کار می‌روند. هم چنین می‌توانید روتین‌های زبان اسمبلی را مستقیماً در کد منبع پاسکال شیئی خود جای دهید؛ برای جزییات بیشتر، فصل۱۳، «کد اسمبلی درون خطی»، را ملاحظه نمایید.

وارد کردن توابع از کتابخانه‌ها

برای وارد کردن[۱] روتین‌ها از یک کتابخانه قابل بارگذاری به طور پویا (DLL. یا so.)، یک دستور به صورت زیر را

```
external stringConstant;
```

به انتهای هدر یک تابع یا روال نرمال ضمیمه کنید، جایی که *stringConstant* نام کتابخانه‌ای میان علامت نقل قول منفرد است. برای مثال، در ویندوز

```
function SomeFunction(S: string): string; external 'strlib.dll';
```

یک تابع با نام *SomeFunction* از strlib.dll وارد می‌کند.

در لینوکس،

```
function SomeFunction(S: string): string; external 'strlib.so';
```

یک تابع به نام *SomeFunction* را از strlib.so وارد می‌کند.

شما می‌توانید یک روتین را تحت نامی متفاوت با آن چه که در کتابخانه دارد وارد کنید. چنان چه این کار را انجام می‌دهید، نام اصلی را در راهنمای **external** مشخص کنید:

```
external stringConstant1 name stringConstant2;
```

جایی که اولین *stringConstant* نام فایل کتابخانه را می‌دهد و دومین *stringConstant* نام اصلی روتین است.

در ویندوز: برای مثال، اعلان زیر یک تابع را از user32.dll (بخشی از API ویندوز) وارد می‌کند.

```
function MessageBox(HWnd: Integer; Text, Caption: PChar; Flags: Integer): Integer;
stdcall; external 'user32.dll' name 'MessageBoxA';
```

---

[۱] Import

نام اصلی تابع *MessageBoxA* است، اما به عنوان *MessageBox* وارد شده است. به جای یک اسم، شما می‌توانید از یک شماره برای شناسایی روتینی که می‌خواهید وارد کنید، استفاده کنید:

```
external stringConstant index integerConstant;
```

جایی که *integerConstant* اندیس روتین در جدول صدور[۱] است.

در لینوکس: برای مثال، اعلان زیر یک تابع استاندارد سیستم را از libc.so.6 وارد می‌کند.

```
function OpenFile(const PathName: PChar; Flags: Integer): Integer; cdecl;
external 'libc.so.6' name 'open';
```

نام اصلی تابع *open* است، اما در این جا به صورت *OpenFile* وارد می‌شود.

در اعلان وارداتی خود، مطمئن شوید که نام روتین از نظر املایی و حالت حروف، دقیقاً همخوان باشد. بعداً، زمانی که شما یک روتین وارد شده را فرامی‌خوانید، نام روتین نسبت به حالت حروف غیرحساس خواهد بود. برای اطلاعات بیشتر درباره کتابخانه‌ها، «کتابخانه‌ها و بسته‌ها» را در فصل۹ ملاحظه نمایید.

سربارگذاری توابع و روال‌ها

شما می‌توانید بیشتر از یک روتین را در دامنه‌ای یکسان با نام یکسان اعلان کنید. این عمل سربارگذاری[۲] خوانده می‌شود. روتین‌های سربارگذاری شده باید با فرمان **overload** اعلان شده باشند و بایستی لیست پارامتر مجزا داشته باشند. برای مثال، به اعلان‌های زیر توجه کنید

```
function Divide(X, Y: Real): Real; overload;
begin
Result := X/Y;
end;
function Divide(X, Y: Integer): Integer; overload;
begin
Result := X div Y;
end;
```

این اعلان‌ها دو تابع، هردو با نام *Divide*، اعلان می‌کنند که پارامترهایی از نوع‌های متفاوت می‌گیرند. هنگامی که *Divide* را فرامی‌خوانید، کامپایلر با نگاه کردن به پارامترهای واقعی ارسال شده در زمان فراخوانی، مشخص می‌کند که کدام تابع فراخوانده شود. برای مثال، Divide(6.0, 3.0) اولین تابع *Divide*

---

[۱] Export table

[۲] Overloading

را فرامی‌خواند، زیرا آرگومان‌هایش مقادیر حقیقی هستند. شما می‌توانید به یک روتین سربارگذاری شده، پارامترهایی را که با نوع‌های هیچ یک از روتین‌های اعلان شده یکسان نیستند، اما با پارامترهای بیشتر از یک اعلان سازگار برای تخصیص باشند، ارسال کنید. این امر بارها پیش می‌آید مثل زمانی که یک روتین با انواع صحیح متفاوت یا انواع حقیقی متفاوت سربارگذاری شده باشد— برای مثال،

```
procedure Store(X: Longint); overload;
procedure Store(X: Shortint); overload;
```

در این گونه موارد، چنان چه انجام بدون ابهام آن ممکن باشد، کامپایلر روتینی را که نوعی از پارامترهایش با کوچکترین دامنه که با پارامترهای واقعی در فراخوان تطبیق می‌کند، احضار خواهد کرد. (به یاد داشته باشید که عبارات ثابت با مقادیر حقیقی همواره از نوع *Extended* هستند.)

روتین‌های سربارگذاری شده بایستی از طریق تعداد پارامترهایی که آنها می‌گیرند یا نوع پارامترهایشان متمایز شوند. از این رو جفت اعلان‌های زیر سبب یک خطای کامپایل می‌شود.

```
function Cap(S: string): string; overload;
...
procedure Cap(var Str: string); overload;
...
```

اما اعلان‌های

```
function Func(X: Real; Y: Integer): Real; overload;
...
function Func(X: Integer; Y: Real): Real; overload;
...
```

مجاز هستند.

هنگامی که یک روتین سربارگذاری شده در یک اعلان **forward** یا واسط (interface) اعلان شده باشد، بایستی تعریف اعلان لیست پارامترهای روتین را تکرار کند. کامپایلر می‌تواند توابع سربارگذاری شده‌ای را که حاوی پارامترهای AnsiString/PChar و WideString/WideChar هستند، از هم متمایز کند. ثابت‌های رشته‌ای یا لیترال‌های ارسال شده به موقعیتی مانند یک موقعیت سربارگذاری شده، به نوع رشته یا کاراکتر خالص ترجمه می‌شوند، یعنی AnsiString/PChar.

```
procedure test(const S: String); overload;
procedure test(const W: WideString); overload;
var
a: string;
b: widestring;
begin
```

```
a := 'a';
b := 'b';
test(a); // calls String version
test(b); // calls WideString version
test('abc'); // calls String version
test(WideString('abc')); // calls widestring version
end;
```

واریانت‌ها نیز می‌توانند به عنوان پارامتر در اعلان توابع سربارگذاری شده به کار برده شوند. واریانت بیشتر از هر نوع ساده‌ای مورد توجه قرارگرفته و متداول‌تر است. پیش از مطابقت واریانت، تقدم و برتری همواره بر مطابقت دقیق نوع داده می‌شود. چنان چه یک واریانت به موقعیت‌هایی مانند یک موقعیت سربارگذاری شده ارسال شود و یک سربار که یک واریانت می‌گیرد در آن موقعیت پارامتر موجود باشد، این وضعیت یک مطابقت دقیق برای نوع Variant در نظر گرفته می‌شود.

این امر می‌تواند سبب برخی اثرات جنبی فرعی با نوع‌های ممیز شناور شود. نوع‌های ممیز شناور از طریق اندازه مطابقت داده می‌شوند. چنان چه در آنجا تطابق دقیقی برای متغیر ممیز شناور ارسال شده به فراخوانی سربارگذاری شده نباشد اما یک پارامتر واریانت در دسترس باشد، واریانت بر هر نوع ممیز شناور کوچکتر مقدم گرفته می‌شود. برای مثال،

```
procedure foo(i: integer); overload;
procedure foo(d: double); overload;
procedure foo(v: variant); overload;
var
v: variant;
begin
foo(1); // integer version
foo(v); // variant version
foo(1.2); // variant version (float literals -> extended precision)
end;
```

این مثال نسخه واریانت از foo، نه نسخه double، را فراخوانی می‌کند، زیرا ثابت 1.2 به طورضمنی یک نوع extended است و extended یک تطابق دقیق برای double نیست. در ضمن Extended یک تطابق دقیق برای واریانت نیست، اما بیشتر واریانت متداول و مورد توجه است (در حالی که double یک نوع کوچکتر از Extended است).

```
foo(Double(1.2));
```

این قالب‌بندی عمل نمی‌کند. در عوض شما بایستی از ثابت‌های نوعدار استفاده کنید.

```
const d: double = 1.2;
begin
foo(d);
end;
```

کد بالا به درستی کار می‌کند و نسخه double را فرامی‌خواند.

```
const s: single = 1.2;
begin
foo(s);
end;
```

در ضمن کد بالا نسخه double از foo را فرامی‌خواند. Single با double جفت و جورتر از واریانت است.

هنگام اعلان مجموعه‌ای از روتین‌های سربارگذاری شده، بهترین راه برای پرهیز از ترفیع ممیز شناور به واریانت این است که به ازای هر نوع ممیز شناور (Single، Double، Extended) نسخه‌ای از تابع سربارگذاری شده را اعلان کنید.

چنان چه از پارامترهای پیش فرض[۱] در روتین‌های سربارگذاری شده استفاده می‌کنید، مواظب اثرات پارامتر دو پهلو[۲] باشید. برای اطلاعات بیشتر، بخش «پارامترهای پیش فرض و روتین‌های سربارگذاری شده» را در همین فصل ملاحظه نمایید.

شما می‌توانید هنگامی که یک روتین سربارگذاری شده را فرامی‌خوانید، اثرات بالقوه سربارگذاری را با قیددار کردن نام روتین محدود نمایید. برای مثال، Unit1.MyProcedure(X, Y) تنها می‌تواند روتین‌های اعلان شده در *Unit1* را فراخوانی کند؛ اگر هیچ روتینی در *Unit1* از نظر اسم و لیست پارامتر با فراخوان مطابقت نداشته باشد، نتیجه یک خطا خواهد بود.

برای آگاهی از اطلاعات بیشتر درباره توزیع متدهای سربارگذاری شده در سلسه مراتب یک کلاس، بخش «سربارگذاری متدها» را در فصل۷ ملاحظه نمایید. به منظور آگاهی از اطلاعاتی درباره صدور روتین‌های سربارگذاری شده از یک کتابخانه اشتراکی، بخش «شرط صادرات» را در فصل۹ ببینید.

---

[۱] Default parameters

[۲] Ambiguous parameter signatures

اعلان‌های محلی

بدنه یک روال یا تابع اغلب با اعلانات متغیرهای محلی استفاده شده در بلوک دستورات روتین، آغاز می‌شود. در ضمن این اعلانات می‌توانند شامل ثابت‌ها، نوع‌ها و روتین‌های دیگر باشند. دامنه یک شناسه محلی به روتین جایی که شناسه اعلان شده است، محدود می‌شود.

روتین‌های تودرتو

برخی اوقات روال‌ها و توابع در میان بخش اعلانات محلی بلوکشان از توابع و روتین‌های دیگر تشکیل می‌شوند. برای مثال، اعلان زیر یک روال با نام *DoSomething* است که از یک روال تورفته تشکیل شده است.

```
procedure DoSomething(S: string);
var
X, Y: Integer;
procedure NestedProc(S: string);
begin
...
end;
begin
...
NestedProc(S);
...
end;
```

دامنه یک روتین تورفته به روال یا تابع، جایی که روتین اعلان شده، محدود می‌شود. در مثال قبلی، *NestedProc* تنها می‌تواند در میان *DoSomething* فراخوانده شود.

به منظور مشاهده مثال‌های واقعی از روتین‌های تودرتو، روال *DateTimeToString*، تابع *ScanDate* و روتین‌های دیگری را که در یونیت *SysUtils* قرار دارند ملاحظه نمایید.

پارامترها

اغلب هدرهای تابع و روال شامل یک لیست پارامتر هستند. برای مثال، در هدر

```
function Power(X: Real; Y: Integer): Real;
```

(X: Real; Y: Integer) لیست پارامتر است.

یک لیست پارامتر دنباله‌ای از اعلانات پارامتر جدا شده توسط نقطه ویرگول و محصور شده در میان پارانتزهاست. هر اعلان یک سری اسامی پارامتر جدا شده توسط ویرگول است که در اغلب موارد بعد

از آنها یک علامت دونقطه و یک شناسه نوع می‌آید و در برخی از موارد با علامت = و یک مقدار پیش فرض پی گرفته می‌شود. اسامی پارامتر باید شناسه‌های معتبری باشند. قبل از هر اعلان می‌تواند یکی از کلمات رزرو شده **var**، **const** و **out** بیاید. مثال‌ها:

```
(X, Y: Real)
(var S: string; X: Integer)
(HWnd: Integer; Text, Caption: PChar; Flags: Integer)
(const P; I: Integer)
```

لیست پارامتر تعداد، ترتیب و نوع پارامترهایی را که باید در زمان فراخوانی روتین به آن ارسال شوند، مشخص می‌کند. اگر یک روتین هیچ پارامتری نگیرد، لیست شناسه‌ها و پارانتزها را از اعلان روتین حذف کنید:

```
procedure UpdateRecords;
begin
...
end;
```

در میان بدنه روال یا تابع، اسامی پارامتر (X و Y در اولین مثال قبلی) می‌توانند همانند متغیرهای محلی مورد استفاده قرار گیرند. اسامی پارامتر را در بخش اعلانات محلی بدنه روال یا تابع اعلان مجدد نکنید.

معناشناسی پارامتر

پارامترها به چند شیوه طبقه‌بندی می‌شوند:

- هر پارامتر به صورت مقداری (*value*)، متغیر (*variable*)، ثابت (*constant*) یا بیرونی (*out*) دسته‌بندی می‌شود. پارامترهای *value* پیش فرض هستند؛ کلمات کلیدی **var**، **const** و **out** به ترتیب بیانگر پارامترهای متغیر، ثابت و بیرونی هستند.
- پارامترهای مقداری (*value*) همواره نوع‌دار هستند، در حالی که پارامترهای ثابت، متغیر و بیرونی یا نوع‌دار هستند یا بدون نوع.
- قواعد مخصوصی به پارامترهای آرایه اعمال می‌شود. بخش «پارامترهای آرایه» را در همین فصل ملاحظه نمایید. فایل‌ها و وهله‌هایی از انواع ساخت‌یافته که حاوی فایل‌ها هستند تنها می‌توانند به صورت پارامترهای متغیر (**var**) ارسال شوند.

پارامترهای متغیر و مقداری

اغلب پارامترها یا پارامترهای مقداری (حالت پیش فرض) هستند یا پارامترهای متغیر (**var**). پارامترهای مقدار به *واسطه مقدار*[۱] ارسال می‌شوند درحالی که پارامترهای متغیر به *واسطه ارجاع*[۲]. برای این که ببینید معنای این حرف چیست، توابع زیر را ملاحظه نمایید.

```
function DoubleByValue(X: Integer): Integer; // X is a value parameter
begin
X := X * 2;
Result := X;
end;
function DoubleByRef(var X: Integer): Integer; // X is a variable parameter
begin
X := X * 2;
Result := X;
end;
```

این توابع نتیجه یکسانی را برمی‌گردانند، اما تنها تابع دومی—*DoubleByRef*—می‌تواند مقدار یک متغیر ارسال شده به خود را تغییر دهد. تصور کنید که تابعی مانند این را فراخوانی می‌کنیم:

```
var
I, J, V, W: Integer;
begin
I := 4;
V := 4;
J := DoubleByValue(I); // J = 8, I = 4
W := DoubleByRef(V); // W = 8, V = 8
end;
```

بعد از اجرا شدن کد، متغیر I، که به *DoubleByValue* ارسال شده بود، همان مقداری را خواهد داشت که ما در ابتدا به آن تخصیص دادیم. اما متغیر V، که به *DoubleByRef* ارسال شده بود، مقدار متفاوتی را خواهد داشت.

یک پارامتر مقداری به صورت یک متغیر محلی که با مقدار ارسال شده در فراخوان تابع یا روال مقداردهی شده، عمل می‌کند. چنان چه یک متغیر را به صورت یک پارامتر مقدار ارسال کنید، روال یا تابع یک کپی از آن را ایجاد می‌کند؛ تغییرات اعمالی بر کپی، اثری بر متغیر اصلی ندارد و زمانی که اجرای برنامه به فراخواننده برمی‌گردد، این تغییرات مفقود می‌شوند.

---

[۱] By value

[۲] By reference

از سوی دیگر، یک پارامتر متغیر به جای یک کپی، مانند یک اشاره‌گر عمل می‌کند. بعد از این که اجرای برنامه به فراخواننده برگردد و اسم پارامتر، خودش به خارج از دامنه رفته باشد، تغییرات اعمالی به پارامتر در میان بدنه یک روال یا تابع کماکان باقی بوده و پابرجا هستند. حتی اگر متغیر یکسانی به دو یا چند پارامتر **var** ارسال شود، هیچ کپی ایجاد نمی‌شود. این امر در مثال زیر شرح داده شده است.

```
procedure AddOne(var X, Y: Integer);
begin
X := X + 1;
Y := Y + 1;
end;
var I: Integer;
begin
I := 1;
AddOne(I, I);
end;
```

بعد از اجرا شدن این کد، مقدار I برابر 3 است.

اگر اعلان یک روتین یک پارامتر **var** را تصریح کند، بایستی یک عبارت قابل تخصیص را —که می‌تواند یک متغیر، ثابت نوع‌دار (در حالت **{$J+}**)، اشاره‌گر برگشت ارجاع داده شده، فیلد یا متغیر اندیس‌دار شده باشد — به یک روتین هنگام فراخوانی آن ارسال کنید. با به کار بردن مثال‌های قبلی که داشتیم، DoubleByRef(7) یک خطا تولید می‌کند، اگر چه DoubleByValue(7) مجاز خواهد بود.

اندیس‌ها و برگشت ارجاعات اشاره‌گر ارسال شده در پارامترهای **var** — برای مثال، DoubleByRef(MyArray[I])— قبل از اجرای روتین، یک بار ارزیابی می‌شوند.

پارامترهای ثابت

یک پارامتر ثابت (**const**) همانند یک ثابت محلی یا یک متغیر فقط-خواندنی است. پارامترهای ثابت با پارامترهای مقدار مشابه هستند، به جز این که شما نمی‌توانید مقداری را به یک پارامتر ثابت در میان بدنه یک روال یا تابع تخصیص دهید، یا این که یک پارامتر ثابت را به صورت یک پارامتر **var** به روتین دیگری ارسال کنید. (اما زمانی که یک ارجاع شیء را به صورت یک پارامتر ثابت ارسال می‌کنید، هنوز هم می‌توانید خاصیت‌های شیء را اصلاح کنید.)

استفاده از **const** به کامپایلر اجازه می‌دهد تا کد را برای پارامترهای نوع ساخت‌یافته و رشته بهینه‌سازی کند. در ضمن یک حفاظ در برابر ارسال ناخواسته یک پارامتر بواسطه ارجاع به روتین

دیگر، آماده می‌کند. برای مثال، در زیر هدر مربوط به تابع *CompareStr* که در یونیت *SysUtils* قرار دارد، نشان داده شده است:

```
function CompareStr(const S1, S2: string): Integer;
```

از آن جایی که S1 و S2 در بدنه *CompareStr* تغییر داده نمی‌شوند، می‌توانند به صورت پارامترهای ثابت اعلان شوند.

پارامترهای بیرونی

یک پارامتر **out**، همانند یک پارامتر متغیر، به واسطه ارجاع (by reference) ارسال می‌شود. گرچه، با یک پارامتر **out**، مقدار اولیه متغیر ارجاع شده توسط روتینی که به آن ارسال می‌شود، دور انداخته می‌شود. پارامتر **out** تنها برای خروجی است؛ یعنی به تابع یا روال می‌گوید کجا خروجی را ذخیره کند، اما هیچ ورودی را فراهم نمی‌کند. برای مثال، به هدینگ روال زیر توجه کنید

```
procedure GetInfo(out Info: SomeRecordType);
```

زمانی که *GetInfo* را فرامی‌خوانید، بایستی متغیری از نوع *SomeRecordType* را به آن ارسال کنید:

```
var MyRecord: SomeRecordType;
...
GetInfo(MyRecord);
```

اما *MyRecord* را برای ارسال هیچ گونه داده‌ای به روال *GetInfo* به کار نمی‌برید؛ *MyRecord* تنها یک محفظه و کانتینری است که می‌خواهید *GetInfo*، اطلاعاتی را که تولید می‌کند، در آن ذخیره کند. فراخوانی برای *GetInfo* قبل از این که کنترل برنامه به روال منتقل شود، بلافاصله حافظه استفاده شده توسط *MyRecord* را آزاد می‌کند.

پارامترهای **out** بارها با مدل‌های شیء توزیع شده[۱] مانند COM و CORBA به کار برده می‌شوند. علاوه بر این، زمانی که یک متغیر مقداردهی نشده را به یک تابع یا روال ارسال می‌کنید، لازم است که از پارامترهای **out** استفاده کنید.

---

[۱] Distributed-object models

پارامترهای بدون نوع

هنگام اعلان پارامترهای **var**، **const** و **out** می‌توانید مشخصات نوع را از قلم بیندازید. (پارامترهـای مقدار باید نوعدار باشند.) برای مثال،

```
procedure TakeAnything(const C);
```

یک روال با نام *TakeAnything* اعلان می‌کند که یک پارامتر از هر نوعی را می‌پذیرد. زمانی کـه چنـین روتینی را فرامی‌خوانید، نمی‌توانید یک ثابت numeral یا عددی بدون نـوع را بـه آن ارسـال کنیـد. در میان بدنه یک تابع یا روال، پارامترهای بدون نوع با همه انواع ناسازگار هستند. بـرای انجـام عملیـات روی یک پارامتر بدون نوع، بایستی آن را تبدیل نوع صریح (قالب‌بندی) نمایید. به طور کلی، کامپـایلر نمی‌تواند بررسی کند که آن اعمال روی پارامترهـای بـدون نـوع معتبـر هسـتند یـا نـه. مثـال زیـر از پارامترهای بدون نوع در یک تابع به نام *Equal* استفاده می‌کند که این تابع تعداد مشخصی از بایت‌های هر دو متغیر را مقایسه می‌کند.

```
function Equal(var Source, Dest; Size: Integer): Boolean;
type
TBytes = array[0..MaxInt - 1] of Byte;
var
N: Integer;
begin
N := 0;
while (N < Size) and (TBytes(Dest)[N] = TBytes(Source)[N]) do
Inc(N);
Equal := N = Size;
end;
```

با اعلانات داده شده زیر

```
type
TVector = array[1..10] of Integer;
TPoint = record
X, Y: Integer;
end;
var
Vec1, Vec2: TVector;
N: Integer;
P: TPoint;
```

می‌توانید فراخوان‌های زیر را به *Equal* انجام دهید:

```
Equal(Vec1, Vec2, SizeOf(TVector))              // compare Vec1 to Vec2
Equal(Vec1, Vec2, SizeOf(Integer) * N)          // compare first N elements of Vec1 and Vec2
Equal(Vec1[1], Vec1[6], SizeOf(Integer) * 5)    // compare first 5 to last 5 elements of Vec1
Equal(Vec1[1], P, 4)                            // compare Vec1[1] to P.X and Vec1[2] to P.Y
```

پارامترهای رشته ای

هنگامی اعلان روتین‌هایی را که پارامترهای رشته کوتاه می‌گیرند، نمی‌توانید تصریح کننده‌های طول را در اعلان پارامتر جای دهید. از این رو، اعلان

```
procedure Check(S: string[20]); // syntax error
```

باعث بروز یک خطای کامپایل می‌شود. اما

```
type TString20 = string[20];
procedure Check(S: TString20);
```

معتبر است. برای اعلان روتینی که پارامترهای رشته کوتاه از طول‌های مختلف می‌گیرد، شناسه ویژه *OpenString* می‌تواند به کار برده شود.

```
procedure Check(S: OpenString);
```

چنان چه راهنماهای کامپایلر **{$H–}** و **{$P+}** هر دو در حال اثر باشند، واژه کلیدی **string** معادل با *OpenString* در اعلانات پارامتر است.

رشته‌های کوتاه، *OpenString*، **$H** و **$P** تنها برای سازگاری با گذشته پشتیبانی می‌شوند. در کدنویسی جدید، شما می‌توانید با استفاده از رشته‌های بلند از این ملاحظات پرهیز کنید.

پارامترهای آرایه

هنگامی که روتین‌هایی را که از پارامترهای آرایه استفاده می‌کنند، اعلان می‌کنید، نمی‌توانید تصریح کننده‌های نوع اندیس را در اعلان پارامتر جای دهید. یعنی، اعلان

```
procedure Sort(A: array[1..10] of Integer); // syntax error
```

باعث بروز یک خطای کامپایل می‌شود. اما

```
type TDigits = array[1..10] of Integer;
procedure Sort(A: TDigits);
```

معتبر است. گرچه برای اغلب مقاصد، *پارامترهای آرایه باز*[۱] راه حل بهتری هستند.

---

[۱] Open array parameters

پارامترهای آرایه باز

پارامترهای آرایه باز به آرایه‌هایی با اندازه‌های مختلف اجازه می‌دهند تا به تابع یا روال یکسانی ارسال شوند. برای تعریف یک روتین با یک پارامتر آرایه باز، از ترکیب نوشتاری/نحوی array of *type* (به جای array[*X*..*Y*] of *type*) در اعلان پارامتر استفاده کنید. برای مثال،

```
function Find(A: array of Char): Integer;
```

یک تابع به نام *Find* اعلان می‌کند که یک آرایه کاراکتر از هر اندازه ای را گرفته و یک عدد صحیح را برمی‌گرداند.

توجه ترکیب نوشتاری/نحوی پارامترهای آرایه باز شبیه ترکیب نوشتاری/نحوی انواع آرایه دینامیک است، اما این دو چیز منظور یکسانی ندارند. مثال پیشین یک تابع ایجاد می‌کند که هر آرایه‌ای از عناصر *Char*، از جمله آرایه‌های دینامیک (اما نه محدود به آنها) را می‌پذیرد. برای اعلان پارامترهایی که می‌بایست آرایه‌های دینامیک باشند، نیازمند تعیین یک شناسه نوع هستید:

```
type TDynamicCharArray = array of Char;
function Find(A: TDynamicCharArray): Integer;
```

برای اطلاعات بیشتر درباره آرایه‌های دینامیک، بخش «آرایه‌های پویا» را در فصل۵ ملاحظه نمایید.

در میان بدنه یک روتین، پارامترهای آرایه باز توسط قواعد زیر کنترل می‌شوند.

- آنها همواره آرایه‌های پایه–صفر هستند. اولین عنصر صفر است، دومی۱ و ... . توابع استاندارد *Low* و *High* به ترتیب صفر و *Length*–1 را برمی‌گردانند. تابع *SizeOf* اندازه آرایه واقعی ارسال شده به روتین را برمی‌گرداند.
- آنها تنها می‌توانند به واسطه عنصر مورد دسترسی باشند. تخصیصات به یک پارامتر آرایه باز دست نخورده مجاز نیست.
- آنها تنها می‌توانند به عنوان پارامترهای آرایه باز یا پارامترهای **var** بدون نوع به روال‌ها و توابع دیگر ارسال شوند. آنها نمی‌توانند به *SetLength* ارسال شوند.

به جای یک آرایه، شما می‌توانید یک متغیر از نوع مبنای پارامتر آرایه باز را ارسال کنید. با این متغیر همانند آرایه‌ای از طول یک رفتار خواهد شد.

زمانی که یک آرایه را به صورت یک پارامتر مقدار آرایه باز ارسال می‌کنید، کامپایلر یک کپی محلی از آرایه را در میان چارچوب پشته روتین ایجاد می‌کند. مواظب باشید که پشته به علت ارسال آرایه‌های بزرگ سرریز نکند.

مثال‌های زیر از پارامترهای آرایه باز برای تعریف یک روال *Clear* استفاده می‌کند که صفر را به هر عنصر واقع در یک آرایه از اعداد حقیقی تخصیص می‌دهد و نیز تابع *Sum* که مجموع عناصر یک آرایه از اعداد حقیقی را محاسبه می‌کند.

```
procedure Clear(var A: array of Real);
var
I: Integer;
begin
for I := 0 to High(A) do A[I] := 0;
end;
function Sum(const A: array of Real): Real;
var
I: Integer;
S: Real;
begin
S := 0;
for I := 0 to High(A) do S := S + A[I];
Sum := S;
end;
```

هر وقت روتین‌هایی را که از پارامترهای آرایه باز استفاده می‌کنند، فراخوانی می‌کنید، می‌توانید سازنده‌های آرایه باز را به آنها ارسال کنید. بخش «سازنده‌های آرایه باز» را در همین فصل ملاحظه نمایید.

پارامترهای آرایه باز واریانت

پارامترهای آرایه باز واریانت به شما اجازه می‌دهند تا آرایه‌ای از عباراتی را که به طور متفاوتی نوع‌دار شده‌اند، به یک تابع یا روال مجزا ارسال کنید. برای تعریف یک روتین با یک پارامتر آرایه باز واریانت، **array of const** را به عنوان نوع پارامتر تعیین کنید. از این رو

```
procedure DoSomething(A: array of const);
```

یک روال به نام *DoSomething* اعلان می‌کند که می‌تواند روی آرایه‌های نامتجانس عمل کند.

ساختار **array of const** معادل با **array of** *TVarRec* است. *TVarRec*، که در یونیت *System* اعلان شده است، یک رکورد با یک بخش واریانت را بیان می‌کند که می‌تواند مقادیری از انواع صحیح، بولی،

کاراکتر، حقیقی، رشته، اشاره‌گر، کلاس، class reference، واسط (interface) و واریانت را نگه دارد. فیلد *VType* از رکورد *TVarRec* نشانگر نوع هر عنصر واقع در آرایه است. برخی از نوع‌ها به جای مقادیر به صورت اشاره‌گر ارسال می‌شوند؛ به خصوص، رشته‌های بلند به صورت *Pointer* ارسال می‌شوند و باید به **string** قالب‌بندی (تبدیل نوع صریح) شوند. برای آگاهی از جزییات بیشتر راهنمای درون خطی را در مورد *TVarRec* ملاحظه نمایید.

مثال زیر از یک پارامتر آرایه باز واریانت در یک تابع استفاده می‌کند که این تابع یک نمایش رشته‌ای از هر عنصر ارسالی به آن را ایجاد کرده و نتیجه را در یک رشته واحد متمرکز می‌کند. روتین‌های دست‌کاری رشته که در این تابع فراخوانده می‌شوند، در *SysUtils* تعریف شده اند.

```
function MakeStr(const Args: array of const): string;
const
BoolChars: array[Boolean] of Char = ('F', 'T');
var
I: Integer;
begin
Result := '';
for I := 0 to High(Args) do
with Args[I] do
case VType of
vtInteger: Result := Result + IntToStr(VInteger);
vtBoolean: Result := Result + BoolChars[VBoolean];
vtChar: Result := Result + VChar;
vtExtended: Result := Result + FloatToStr(VExtended^);
vtString: Result := Result + VString^;
vtPChar: Result := Result + VPChar;
vtObject: Result := Result + VObject.ClassName;
vtClass: Result := Result + VClass.ClassName;
vtAnsiString: Result := Result + string(VAnsiString);
vtCurrency: Result := Result + CurrToStr(VCurrency^);
vtVariant: Result := Result + string(VVariant^);
vtInt64: Result := Result + IntToStr(VInt64^);
end;
end;
```

ما می‌توانیم این تابع را با استفاده از یک سازنده آرایه باز[۱]، فراخوانی کنیم (بخش «سازنده‌های آرایه باز» را در همین فصل ملاحظه نمایید). برای مثال،

```
MakeStr(['test', 100, ' ', True, 3.14159, TForm])
```

رشته "test100 T3.14159TForm" را برمی‌گرداند.

---

[۱] Open array constructor

پارامترهای پیش فرض

شما می‌توانید مقادیر پارامتر پیش فرض را در هدینگ یک تابع یا روال معین کنید. مقادیر پیش فرض تنها برای پارامترهای نوع‌دار **const** و مقداری مجاز هستند. به منظور فراهم سازی یک مقدار پیش فرض، اعلان پارامتر را با علامت = که بعد از آن یک عبارت ثابت سازگار برای تخصیص با نوع پارامتر می‌آید، خاتمه دهید. برای مثال، با اعلان‌های داده شده زیر

```
procedure FillArray(A: array of Integer; Value: Integer = 0);
```

فراخوان برای روال‌های زیر معادل هستند.

```
FillArray(MyArray);
FillArray(MyArray, 0);
```

یک اعلان پارامتر چندگانه نمی‌تواند یک مقدار پیش فرض را تعیین کند. از این رو، درحالی که

```
function MyFunction(X: Real = 3.5; Y: Real = 3.5): Real;
```

مجاز است،

```
function MyFunction(X, Y: Real = 3.5): Real; // syntax error
```

مجاز نیست.

پارامترها با مقادیر پیش فرض بایستی در انتهای لیست پارامتر ظاهر شوند و معنای این حرف، این است که همه پارامترها بعد از اولین مقدار پیش فرض اعلان شده نیز باید مقادیر پیش فرض داشته باشند.

```
procedure MyProcedure(I: Integer = 1; S: string); // syntax error
```

مقادیر پیش فرض تعیین شده در یک نوع رویه‌ای آنهایی را که در یک روتین واقعی تعیین شده‌اند، باطل می‌کنند. از این رو، با اعلان‌های داده شده

```
type TResizer = function(X: Real; Y: Real = 1.0): Real;
function Resizer(X: Real; Y: Real = 2.0): Real;
var
F: TResizer;
N: Real;
```

دستورات

```
F := Resizer;
F(N);
```

منجر به ارسال مقادیر (*N*, 1.0) به *Resizer* می‌شوند.

پارامترهای پیش فرض به مقادیری محدود می‌شوند که می‌توانند بوسیله یک عبارت ثابت تعیین شوند. (بخش «عبارات ثابت» را در فصل۵ ملاحظه نمایید.) از این رو پارامترهایی از نوع یک آرایه پویا، رویه‌ای، کلاس، ارجاع کلاس یا واسط (interface) نمی‌توانند هیچ مقداری به غیر از **nil** را به عنوان مقادیر پیش فرض خود داشته باشند. پارامترهایی از نوع یک رکورد، واریانت، فایل، آرایه استاتیک یا شیء به هیچ وجه نمی‌توانند مقدار پیش فرض داشته باشند. برای آگاهی از اطلاعات بیشتر درباره فراخوانی روتین‌ها با مقادیر پارامتر پیش فرض، بخش «فراخوانی توابع و روال‌ها» را در همین فصل ملاحظه نمایید.

پارامترهای پیش فرض و روتین‌های سربارگذاری شده

چنان چه مقادیر پارامتر پیش فرض را در یک روتین سربارگذاری شده به کار می‌برید، از اثرات پارامتر مبهم پرهیز کنید. برای مثال به کد زیر را ملاحظه کنید.

```
procedure Confused(I: Integer); overload;
...
procedure Confused(I: Integer; J: Integer = 0); overload;
...
Confused(X);                    // Which procedure is called?
```

در حقیقت، هیچ کدام از دو روال بالا فراخوان نمی‌شوند. این کد یک خطای کامپایل تولید می‌کند.

پارامترهای پیش فرض در اعلان‌های forward و interface

اگر روتینی یک اعلان **forward** داشته باشد یا در بخش واسط (**interface**) یک یونیت رخ دهد، مقادیر پارامتر پیش فرض باید در اعلان **forward** یا واسط تعیین شوند. در این حالت، مقادیر پیش فرض می‌توانند از تعریف اعلان (implementation) حذف شوند؛ اما اگر تعریف اعلان حاوی مقادیر پیش فرض باشد، این مقادیر باید دقیقاً با آنهایی که در اعلان **forward** یا interface هستند، مطابقت داشته باشند.

فراخوانی روال‌ها و توابع

هر وقت یک تابع یا روال را فرامی‌خوانید، کنترل برنامه از آن نقطه جایی که فراخوان انجام شده، به بدنه روتین منتقل می‌شود. شما می‌توانید فراخوانی را با استفاده از نام اعلان شده روتین (با یا بدون

توصیف کننده‌ها) یا استفاده از یک متغیر رویه‌ای که به آن روتین اشاره می‌کند، انجام دهید. در هر دو حالت، چنان چه روتین با پارامترهایی اعلان شده است، فراخوان شما به آن روتین باید پارامترهایی را که در ترتیب و نوع با لیست پارامترهای روتین منطبق هستند، ارسال کند. پارامترهایی که به یک روتین ارسال می‌کنید، *پارامترهای واقعی* خوانده می‌شوند، در حالی که پارامترهای واقع در اعلان روتین، *پارامترهای صوری* خوانده می‌شوند.

هنگام فرخوانی یک روتین، به یاد داشته باشید که

- عبارات به کار رفته برای ارسال پارامترهای مقدار و **const** نوع‌دار باید سازگار برای تخصیص با پارامترهای صوری متناظر باشند.
- عبارات به کار رفته برای ارسال پارامترهای **var** و **out** باید به طور همسان با پارامترهای صوری متناظر نوع‌دار شده باشند، مگر این که پارامترهای صوری بدون نوع باشند.
- تنها عبارات قابل تخصیص می‌توانند برای ارسال به پارامترهای **var** و **out** به کار روند.

چنان چه پارامترهای صوری یک روتین، بدون نوع باشند، ثابت‌های عددی[۱] و ثوابت صحیح[۲] با مقادیر عددی نمی‌توانند به عنوان پارامترهای واقعی به کار برده شوند.

هنگام فراخوانی یک روتین که از مقادیر پارامتر پیش فرض استفاده می‌کند، همه پارامترهای واقعی که بعد از اولین پارامتر مقدار پیش فرض گرفته می‌آیند نیز باید از مقادیر پیش فرض استفاده کنند؛ فراخوان‌هایی به فرم SomeFunction(, , X) مجاز نیستند.

زمانی که همه و تنها پارامترهای پیش فرض را به یک روتین ارسال می‌کنید، می‌توانید از پارانتزها صرف‌نظر کنید. برای مثال، با روال داده شده زیر

```
procedure DoSomething(X: Real = 1.0; I: Integer = 0; S: string = '');
```

فراخوان‌های زیر معادل هستند.

```
DoSomething();
DoSomething;
```

---

[۱] Numerals

[۲] True constants

سازنده‌های آرایه باز

سازنده‌های آرایه باز به شما امکان می‌دهند تا آرایه‌ها را مستقیماً در میان فراخوان‌های تابع و روال ایجاد کنید. آنها تنها می‌توانند به صورت پارامترهای آرایه باز یا پارامترهای آرایه باز واریانت ارسال شوند.

یک سازنده آرایه باز، مانند یک سازنده مجموعه، دنباله‌ای از عبارات جداشده بوسیله ویرگول است که در میان براکت‌ها (**[ ]**) محصور می‌شوند. برای مثال، با اعلان‌های داده شده زیر

```
var I, J: Integer;
procedure Add(A: array of Integer);
```

شما می‌توانید با دستور زیر، روال *Add* را فراخوانی کنید

```
Add([5, 7, I, I + J]);
```

این کد معادل با کد زیر است

```
var Temp: array[0..3] of Integer;
...
Temp[0] := 5;
Temp[1] := 7;
Temp[2] := I;
Temp[3] := I + J;
Add(Temp);
```

سازنده‌های آرایه باز تنها می‌توانند به عنوان پارامترهای مقداری یا **const** ارسال شوند. عبارات واقع در یک سازنده باید با نوع مبنای پارامتر آرایه سازگار برای تخصیص باشند. در مورد یک پارامتر آرایه باز واریانت، عبارات می‌توانند از نوع‌های متفاوتی باشند.

فصل ۷

کلاس‌ها و اشیاء

یک کلاس یا نوع کلاس، ساختاری را که از فیلدها[۱]، متدها[۲] و خاصیت‌ها[۳] تشکیل شده، تعریف می‌کند. وهله‌های یک نوع کلاس اشیاء[۴] خوانده می‌شوند. فیلدها، متدها و خواص یک کلاس اجزا یا اعضای آن کلاس خوانده می‌شوند.

- یک فیلد در اصل متغیری است که بخشی از یک شیء است. مانند فیلدهای یک رکورد، فیلدهای یک کلاس آیتم‌های داده‌ای را که در هر وهله از کلاس وجود دارند، بیان می‌کنند.
- متد، یک روال یا تابع مرتبط شده با یک کلاس است. اغلب متدها روی اشیاء — یعنی، وهله‌های یک کلاس— عمل می‌کنند. برخی متدها (که متدهای کلاسی خوانده می‌شوند) خود روی انواع کلاس عمل می‌کنند.
- خاصیت، یک واسطه به داده‌های مرتبط با یک شیء است (که اغلب در یک فیلد ذخیره می‌شود). خاصیت‌ها دارای تصریح کننده‌های دسترسی هستند، که چگونگی خواندن و اصلاح

---

۱ Fields

۲ Methods

۳ Properties

۴ Objects

داده آنها را تعیین می‌کنند. در اغلب موارد، در بخش‌های دیگر برنامه یک خاصیت — بیرون از خود شیء — مانند یک فیلد ظاهر می‌شود.

اشیاء بلوک‌هایی از حافظه هستند که به طور دینامیک تخصیص حافظه شده‌اند و ساختار آنها توسط نوع کلاس‌شان مشخص می‌شود. هر شیء یک کپی منحصر به فرد از هر فیلد تعریف شده در کلاس دارد، اما همه وهله‌های یک کلاس متدهای یکسانی را به اشتراک می‌گذارند. اشیاء توسط متدهای ویژه‌ای که سازنده[۱] و تخریب کننده[۲] خوانده می‌شوند، ایجاد شده و از بین می‌روند.

یک متغیر از یک نوع کلاس در واقع اشاره‌گری است که به یک شیء اشاره می‌کند. از این رو بیشتر از یک متغیر می‌توانند به شیء یکسانی ارجاع کنند. مشابه اشاره‌گرهای دیگر، متغیرهای نوع کلاس می‌توانند مقدار **nil** را نگه دارند. اما نباید به طور ضمنی یک متغیر نوع کلاس را برای دسترسی به شیئی که به آن اشاره می‌کند، برگشت ارجاع کنید. برای مثال، SomeObject.Size := 100 مقدار 100 را به خاصیت *Size* شیء ارجاع شده توسط *SomeObject* تخصیص می‌دهد؛ همان کد را به صورت SomeObject^.Size := 100 ننویسید.

انواع کلاس

یک نوع کلاس قبل از این که نمونه‌سازی شود، باید اعلان شده و یک نام به آن داده شود. (شما نمی‌توانید یک نوع کلاس را در مابین یک اعلان متغیر تعریف کنید.) کلاس را تنها در بیرونی‌ترین دامنه یک برنامه یا یونیت اعلان کنید، نه در اعلان یک تابع یا روال. اعلان یک نوع کلاس قالب زیر را دارد

```
type className = class (ancestorClass)
memberList
end;
```

جایی که *className* هر گونه شناسه معتبری بوده، (*ancestorClass*) اختیاری است و *memberList* اعضای کلاس — یعنی فیلدها، متدها و خاصیت‌ها— را اعلان می‌کند. اگر از (*ancestorClass*) صرف‌نظر کنید، در این صورت کلاس جدید مستقیماً از کلاس از پیش تعریف شده *TObject* ارث می‌برد. چنان چه (*ancestorClass*) را ضمیمه اعلان کنید و *memberList* تهی باشد، شما می‌توانید که

---

[۱] Constructor

[۲] Destructor

از end صرفنظر کنید. در ضمن اعلان یک نوع کلاس می‌تواند دربردارنده لیستی از واسط‌های پیاده‌سازی شده توسط کلاس باشد؛ بخش «پیاده‌سازی واسط‌ها» را در فصل ۱۰ ملاحظه نمایید. متدها در اعلان یک کلاس به صورت هدینگ روال‌ها و توابع ظاهر می‌شوند، بدون هیچ بدنه‌ای. تعریف اعلان مربوط به هر متد در جای دیگری از برنامه ظاهر می‌شود. برای مثال، در زیر اعلان کلاس *TMemoryStream* از یونیت *Classes* آورده شده است.

```
type
TMemoryStream = class(TCustomMemoryStream)
private
FCapacity: Longint;
procedure SetCapacity(NewCapacity: Longint);
protected
function Realloc(var NewCapacity: Longint): Pointer; virtual;
property Capacity: Longint read FCapacity write SetCapacity;
public
destructor Destroy; override;
procedure Clear;
procedure LoadFromStream(Stream: TStream);
procedure LoadFromFile(const FileName: string);
procedure SetSize(NewSize: Longint); override;
function Write(const Buffer; Count: Longint): Longint; override;
end;
```

TMemoryStream (واقع در یونیت *Classes*) با ارث بردن اغلب اعضای خود از *TStream*، از آن مشتق می‌شود. اما *TMemoryStream* متدها و خواص متعددی، از جمله متد تخریب کننده‌اش، یعنی *Destroy*، را تعریف— یا تعریف مجدد— می‌کند. سازنده *TMemoryStream* یعنی *Create*، بدون تغییر از *TObject* به ارث رسیده و از این رو اعلان مجدد نمی‌شود. هر عضو به صورت خصوصی[۱]، حفاظت‌شده[۲] یا عمومی[۳] اعلان می‌شود (این کلاس اعضای منتشر شده[۴] ندارد)؛ برای آگاهی از معانی این قیود، بخش «دامنه اعضای کلاس» را در همین فصل ملاحظه نمایید.

با اعلانات داده شده قبلی، می‌توانید یک وهله از *TMemoryStream* را به صورت زیر ایجاد نمایید:

```
var stream: TMemoryStream;
stream := TMemoryStream.Create;
```

---

۱ Private

۲ Protected

۳ Public

۴ Published

وراثت و دامنه

زمانی که یک کلاس را اعلان می‌کنید، شما می‌توانید بلافاصله نیاکان این کلاس را تعیین کنید. برای مثال،

```
type TSomeControl = class(TControl);
```

یک کلاس به نام *TSomeControl* اعلان می‌کند که از *TControl* مشتق می‌شود. یک نوع کلاس به طور خودکار همه اعضای خود را از نیای بی واسطه‌اش ارث می‌برد. هر کلاس می‌تواند اعضای جدیدی را اعلان کرده و عناصر به ارث رسیده را تعریف مجدد کند، اما یک کلاس نمی‌تواند اعضای تعریف شده در یک نیا را حذف کند. از این رو *TSomeControl* از همه اعضای تعریف شده در *TControl* و هر یک از نیاکان *TControl* تشکیل می‌شود.

دامنه شناسه یک عضو از یک نقطه جایی که عضو اعلان شده، شروع می‌شود و تا انتهای اعلان کلاس ادامه می‌یابد و در سرتاسر فرزندان کلاس و بلوک‌های همه متدهای تعریف شده در کلاس و فرزندانش بسط می‌یابد.

TClass و TObject

کلاس *TObject* که در یونیت *System* اعلان شده است، نیای نهایی (یا پایه) همه کلاس‌های دیگر است. *TObject* تنها یک مشت متد، از جمله یک سازنده و یک تخریب کننده پایه را تعریف می‌کند. علاوه بر *TObject*، یونیت *System* نوع ارجاع کلاس *TClass* را اعلان می‌کند:

```
TClass = class of TObject;
```

به منظور آگاهی بیشتر از اطلاعات بیشتر درباره انواع ارجاع کلاس، بخش «منابع کلاس» را در همین فصل ملاحظه نمایید.

اگر اعلان یک نوع کلاس یک نیا را تعیین نکند، کلاس مستقیماً از *TObject* ارث می‌برد. از این رو

```
type TMyClass = class
...
end;
```

معادل است با

```
type TMyClass = class(TObject)
...
```

```
end;
```

به علت خوانایی، قالب دوم بیشتر توصیه می‌شود.

سازگاری انواع کلاس

یک نوع کلاس سازگار برای تخصیص با نیاکان خود است. از این رو متغیری از یک نوع کلاس می‌تواند به یک وهله از هر نوع فرزند ارجاع (یا اشاره) کند. برای مثال، با اعلان‌های داده شده زیر

```
type
TFigure = class(TObject);
TRectangle = class(TFigure);
TSquare = class(TRectangle);
var
Fig: TFigure;
```

متغیر *Fig* می‌تواند با مقادیری از نوع *TFigure*، *TRectangle* و *TSquare* تخصیص داده شود.

انواع شیء

به عنوان یک جایگزین برای انواع کلاس، شما می‌توانید *انواع* شیء را با استفاده از ترکیب نوشتاری/نحوی زیر اعلان کنید

```
type objectTypeName = object (ancestorObjectType)
memberList
end;
```

جایی که *objectTypeName* هر شناسه معتبری بوده و (*ancestorObjectType*) اختیاری است و *memberList* فیلدها، متدها و خصوصیت‌ها را اعلان می‌کند. چنان چه از (*ancestorObjectType*) صرف‌نظر شود، در این صورت نوع جدید هیچ نیایی نخواهد داشت. انواع شیء نمی‌توانند دارای اعضای منتشر شده (published) باشند.

از آن جایی که انواع شیء از شیء *TObject* مشتق نمی‌شوند، هیچ سازنده، تخریب کننده یا متد توکار دیگری را ارائه نمی‌دهند. شما می‌توانید با استفاده از روال *New* وهله‌هایی از یک نوع شیء را ایجاد کرده و آنها را با روال *Dispose* تخریب کنید یا خیلی ساده‌تر، می‌توانید متغیرهایی از یک نوع شیء را اعلان نمایید، درست همان طور که برای رکوردها این کار را انجام می‌دهید. انواع شیء تنها برای سازگاری با گذشته پشتیبانی می‌شوند. استفاده از آنها توصیه نمی‌شود.

میدان دید اعضای کلاس

هر عضو یک کلاس مشخصه‌ای به نام *میدان دید*[۱] دارد، که با یکی از کلمات رزرو شده **private**، **protected**، **public**، **published** یا **automated** نشان داده می‌شود. برای مثال،

```
published property Color: TColor read GetColor write SetColor;
```

یک خاصیت published (منتشر شده) به نام *Color* اعلان می‌کند. میدان دید تعیین می‌کند که کجا و چگونه یک عضو می‌تواند قابل دسترس باشد، با خصوصی (private) به نمایندگی از کمترین قابلیت دسترسی، حفاظت شده (protected) به نمایندگی از یک تراز میانه از قابلیت دسترسی و عمومی (public)، منتشر شده (published) و automated به نمایندگی از بزرگترین قابلیت دسترسی.

اگر اعلان یک عضو بدون تصریح کننده میدان دیدش ظاهر شود، این عضو همان میدان دیدی را دارد که عضو مقدم بر آن دارد. اعضای واقع در ابتدای اعلان یک کلاس که میدان دید مشخصی ندارند، به طور پیش فرض منتشرشده (published) هستند، به شرط آن که کلاس در حالت **{$M+}** کامپایل شود یا از یک کلاس کامپایل شده در حالت **{$M+}** مشق شود؛ در غیر این صورت، چنین اعضایی عمومی (public) هستند.

به منظور خوانایی بیشتر کد، بهتر است که یک اعلان کلاس توسط میدان دید (سطح دسترسی) سازماندهی شود، با قراردادن همه اعضای خصوصی (private) باهم و بعد از آنها همه اعضای حفاظت شده (protected) و الی آخر. در این روش هر یک از واژه‌های کلیدی میدان دید، حداکثر یک بار ظاهر شده و ابتدای یک بخش جدید از اعلان را علامت‌گذاری می‌کند. بنابراین یک اعلان کلاس نوعی بایستی به این شکل باشد:

```
type
TMyClass = class(TControl)
private
...          { private declarations here}
protected
...          { protected declarations here }
public
...          { public declarations here }
published
```

---

[۱] Visibility

```
...          { published declarations here }
end;
```

شما می‌توانید سطح دسترسی یک عضو را در یک کلاس فرزند با اعلان مجدد آن افزایش دهید، اما سطح دسترسی آن را نمی‌توانید کاهش دهید. برای مثال، یک خاصیت حفاظت شده (protected) واقع در یک فرزند می‌تواند عمومی(public) شود، اما خصوصی (private) نه. به علاوه، اعضای منتشر شده (published) واقع در یک کلاس فرزند نمی‌توانند عمومی(public) شوند. برای اطلاعات بیشتر بخش «لغو و اعلان مجدد خاصیت» را در همین فصل ملاحظه نمایید.

اعضای public، private و protected

یک عضو خصوصی (private) خارج از یونیت یا برنامه، جایی که کلاسش اعلان شده، غیر قابل مشاهده است. به عبارت دیگر، یک متد خصوصی (private) نمی‌تواند از مدول دیگری فراخوانده شود و یک فیلد یا خاصیت خصوصی (private) از مدول دیگری نمی‌تواند خوانده شده یا تغییر داده شود. با قرار دادن اعلان‌های کلاس مربوطه در مدول یکسان، شما می‌توانید به کلاس‌ها امکان دسترسی به اعضای خصوصی (private) یک کلاس دیگر را بدهید، بدون این که این اعضا را از نظر دسترسی گسترده‌تر سازید.

یک عضو حفاظت شده (protected) در هرجایی از مدول، جایی که کلاسش اعلان شده و از هر کلاس فرزند، قابل مشاهده است، قطع نظر از مدول، جایی که کلاس فرزند ظاهر می‌شود. به عبارت دیگر، از تعریف هر متد متعلق به یک کلاس که از یک کلاس که عضو حفاظت شده (protected) در آنجا اعلان شده مشتق شده باشد، یک متد حفاظت شده (protected) می‌تواند فراخوانی شود و یک فیلد یا خاصیت حفاظت شده (protected) خوانده یا تغییر داده شود. اعضایی که صرفاً نامزد استفاده شدن در پیاده‌سازی کلاس‌های مشتق شده هستند، معمولاً حفاظت شده (protected) می‌باشند.

یک عضو عمومی (public) از هرجایی که کلاسش بتواند مورد ارجاع (یا اشاره) باشد، قابل مشاهده است.

اعضای منتشر شده (published)

اعضای منتشرشده (published) همان میدان دیدی را دارند که اعضای عمومی (public) دارنـد. تفـاوت در این است که **اطلاعات زمان اجرا** (RTTI) برای اعضای منتشرشده (published) تولید می‌شـود. RTTI

به یک برنامه اجازه می‌دهد تا فیلدها و خواص یک شیء را به طور دینامیک جستجو کرده و متدهایش را مکان‌یابی کند. RTTI هنگام ذخیره‌سازی و بارگذاری از طریق فایل‌ها، برای دسترسی به مقادیر خواص به کار برده می‌شود تا خواص را در ناظر شیء[۱] نمایش داده و متدهای ویژه‌ای (به نام گرداننده رویداد[۲]) را با خواص ویژه‌ای (به نام رویداد[۳]) همبسته سازد.

اعضای منتشرشده (published) محدود به انواع داده معینی هستند. انواع ترتیبی، رشته، کلاس، واسط و اشاره‌گر–متد می‌توانند منتشرشده (published) گردند. به این ترتیب شما می‌توانید نوع را با **published** تنظیم کنید، به شرط این که مرزهای بالایی و پایینی نوع مبنا مقادیر ترتیبی بین صفر تا ۳۱ داشته باشند. (به عبارت دیگر، مجموعه بایستی در یک بایت، کلمه یا کلمه مضاعف بگنجد.) هر نوع حقیقی به استثنای *Real48* می‌تواند منتشرشده (published) گردد. خاصیت‌های یک نوع آرایه (در زیرخاصیت‌های آرایه به طور مجزا مورد بحث قرار می‌گیرند) نمی‌توانند منتشرشده (published) باشند.

برخی از خاصیت‌ها، با این که منتشر شده (published) هستند، به طور کامل توسط سیستم جریان‌داری[۴] پشتیبانی نمی‌شوند. این‌ها شامل خاصیت‌هایی از انواع رکورد، خاصیت‌های آرایه از همه انواع قابل انتشار و خاصیت‌هایی از نوع شمارشی که حاوی مقادیر بی‌نام هستند (بخش «انواع شمارشی با رتبه‌های تخصیص یافته به طور صریح» را در فصل۵ ملاحظه نمایید)، می‌باشند. چنان چه خاصیتی از این نوع را منتشر کنید، ناظر شیء (Object Inspector) دلفی آن را به درستی نمایش نخواهد داد، یا زمانی که اشیاء به سمت دیسک جاری شوند مقدار خاصیت حفظ نخواهد شد.

همه متدها قابل انتشار هستند، اما یک کلاس نمی‌تواند دو یا چند متد سربارگذاری شده را با نام یکسان، منتشر کند. فیلدها تنها در صورتی می‌توانند منتشرشده (published) باشند که از یک نوع کلاس یا واسط باشند.

یک کلاس نمی‌تواند اعضای منتشرشده (published) داشته باشد مگر این که درحالت **{+$M}** کامپایل شده باشد یا از یک کلاس کامپایل شده در حالت **{+$M}** مشتق شده باشد. اغلب کلاس‌ها با اعضای

---

۱ Object Inspector

۲ Event handler

۳ Event

۴ Streaming system

منتشرشده (published) از *TPersistent* مشتق می‌شوند که در حالت **{$M+}** کامپایل شده است، بنابراین به ندرت لازم می‌شود که از فرمان **$M** استفاده کرد.

اعضای Automated

اعضای automated همان سطح دسترسی (میدان دید) را دارند که اعضای عمومی (public) دارند. تفاوت در این است که اطلاعات نوع اتوماسیون[۱] (مورد نیاز سرورهای اتوماسیون) برای اعضای automated تولید می‌شوند. اعضای automated نوعاً تنها در کلاس‌های ویندوز ظاهر می‌شوند و برای برنامه‌نویسی لینوکس توصیه نمی‌شوند. واژه کلیدی **automated** برای سازگاری با گذشته پشتیبانی می‌شود. کلاس *TAutoObject* واقع در یونیت *ComObj* از **automated** استفاده نمی‌کند.

محدودیت‌های زیر به متدها و خواص اعلان شده به عنوان automated اعمال می‌شوند.

- انواع همه خاصیت‌ها، پارامترهای خاصیت آرایه، پارامترهای متد و نتایج توابع باید automatable باشند (یعنی قابل automated شدن باشند). نوع‌های *Byte*، *Currency*، *Real*، *Double*، *Longint*، *Integer*، *Single*، *Smallint*، *AnsiString*، *WideString*، *TDateTime*، *Variant*، *OleVariant*، *WordBool* و همه انواع واسط (interface)، انواع قابل automated شدن هستند.
- اعلان‌های متد باید از قرارداد فراخوان پیش فرض **register** استفاده کنند. آنها می‌توانند مجازی[۲] باشند اما دینامیک[۳] نه.
- اعلان‌های خاصیت می‌توانند دارای تصریح کننده‌های دسترسی (**read** و **write**) باشند اما تصریح کننده‌های دیگر (**index**، **stored**، **default** و **nodefault**) مجاز نیستند. تصریح کننده‌های دسترسی بایستی شناسه یک متد را لیست کنند که از قرارداد فراخوانی **register** استفاده کند؛ شناسه‌های فیلد مجاز نیستند.

اعلان‌های خاصیت باید یک نوع را تصریح کنند. ابطال‌ها و تعریف مجدد خاصیت مجاز نیستند.

---

۱ Automation type information

۲ Virtual

۳ Dynamic

اعلان یک متد یا خاصیت automated می‌تواند دارای یک فرمان **dispid** باشد. تعیین یک ID از قبل استفاده شده در یک فرمان **dispid** سبب بروز یک خطا می‌شود.

در ویندوز، این راهنما باید با یک ثابت عددی صحیح که یک ID ارسال اتوماسیون برای عضو تعیین می‌کند، پی گرفته شود. در غیر این صورت، کامپایلر به طور خودکار یک ID ارسال به عضو تخصیص می‌دهد که یک واحد بزرگتر از بزرگترین ID ارسال به کار رفته توسط هر متد یا خاصیت در کلاس و نیاکانش است. برای آگاهی از اطلاعات بیشتر درباره اتوماسیون (تنها در ویندوز)، بخش «اشیاء اتوماسیون (تنها در ویندوز)» را در فصل ۱۰ ملاحظه نمایید.

اعلان‌های Forward و کلاس‌های وابسته متقابل

اگر اعلان یک نوع کلاس با کلمه **class** و یک نقطه ویرگول خاتمه یابد — یعنی قالب زیر را داشته باشد

```
type className = class;
```

با هیچ نیا یا اعضای کلاسی که بعد از کلمه **class** لیست شده باشد— در این صورت یک اعلان پیشرو[1] خواهیم داشت. یک اعلان پیشرو باید توسط یک تعریف اعلان از همان کلاس در میان همان بخش اعلان نوع قطعی شود. به عبارتی دیگر، میان یک اعلان پیشرو و تعریف اعلانش، هیچ چیزی به استثنای اعلانات نوع دیگر نمی‌تواند ظاهر شود. اعلان‌های پیشرو (forward) اجازه کلاس‌های وابسته متقابل را می‌دهند. برای مثال،

```
type
TFigure = class;     // forward declaration
TDrawing = class
Figure: TFigure;
...
end;
TFigure = class      // defining declaration
Drawing: TDrawing;
...
end;
```

اعلان‌های پیشرو (forward) را با اعلان‌های کامل انواعی که از *TObject* بدون اعلان هر یک از اعضای کلاس، مشتق می‌شوند، اشتباه نگیرید.

---

[1] forward

```
type
TFirstClass = class;              // this is a forward declaration
TSecondClass = class              // this is a complete class declaration
end;                              //
TThirdClass = class(TObject);  // this is a complete class declaration
```

فیلدها

یک فیلد شبیه متغیری است که به یک شیء تعلق دارد. فیلدها می‌توانند از هر نوعی، از جمله انواع کلاس، باشند. (یعنی، فیلدها می‌توانند منابع شیء را نگه‌داری کنند.) فیلدها معمولاً خصوصی (private) هستند.

به منظور تعریف یک عضو فیلدی کلاس، به سادگی فیلد را اعلان نمایید همان طور که برای یک متغیر این کار را می‌کنید. تمامی اعلانات فیلد بایستی قبل از هر اعلان خاصیت یا متدی اتفاق بیفتند. برای مثال، اعلان زیر یک کلاس به نام *TNumber* اعلان می‌کند که تنها عضوش ، غیر از متدها که از *TObject* به ارث می‌رسد، یک فیلد عدد صحیح به نام *Int* است.

```
type TNumber = class
Int: Integer;
end;
```

فیلدها به طور استاتیک مرزبندی می‌شوند؛ یعنی ارجاعات به آنها در زمان کامپایل ثابت هستند. برای درک معنای این حرف، کد زیر را ملاحظه کنید.

```
type
TAncestor = class
Value: Integer;
end;
TDescendant = class(TAncestor)
Value: string;                          // hides the inherited Value field
end;
var
MyObject: TAncestor;
begin
MyObject := TDescendant.Create;
MyObject.Value := 'Hello!';              // error
TDescendant(MyObject).Value := 'Hello!';    // works!
end;
```

با این که *MyObject* یک وهله *TDescendant* را نگه می‌دارد، به عنوان *TAncestor* اعلان شده است. از این رو کامپایلر *MyObject.Value* را به عنوان یک استناد به فیلد (integer) اعلان شده در *TAncestor*

تعبیر می‌کند گرچه هر دو فیلد در شیء *TDescendant* موجود هستند؛ *Value* موروثی توسط *Value* جدیدتر مخفی شده است و می‌تواند از طریق یک قالب‌بندی (تبدیل نوع صریح) در دسترس باشد.

متدها

متد، یک تابع یا روال مرتبط با یک کلاس است. یک فرخوان به یک متد، یک شیء (یا، چنان چه یک متد کلاسی باشد، کلاس) را تعیین می‌کند که متد باید روی آن عمل کند. برای مثال،

```
SomeObject.Free
```

متد *Free* را در *SomeObject* فراخوانی می‌کند.

اعلان‌ها و پیاده‌سازی‌های متد

در میان یک اعلان کلاس، متدها به صورت هدینگ توابع و روال‌ها ظاهر می‌شوند، که مانند اعلانهای پیشرو (**forward**) عمل می‌کنند. یک جایی بعد از اعلان کلاس، اما در میان همان مدول، هر متد بایستی توسط یک تعریف اعلان پیاده‌سازی شود. برای مثال، تصور کنید که اعلان *TMyClass* شامل یک متد به نام *DoSomething* باشد:

```
type
TMyClass = class(TObject)
...
procedure DoSomething;
...
end;
```

تعریف اعلان مربوط به *DoSomething* باید بعداً در مدول ظاهر شود:

```
procedure TMyClass.DoSomething;
begin
...
end;
```

در حالی که یک کلاس می‌تواند در هر یک از بخش‌های پیاده‌سازی (**implementation**) یا واسط (**interface**) یک یونیت ظاهر شود، تعریف اعلان متدهای یک کلاس باید در بخش پیاده‌سازی (**implementation**) جای داشته باشند.

در هدینگ یک تعریف اعلان، اسم متد همواره با اسم کلاسی که متد به آن تعلق دارد قیددار می‌شود. سرفصل می‌تواند لیست پارامتر اعلان کلاس را تکرار کند؛ چنان چه این کار انجام شود، ترتیب، نوع و

اسامی پارامترها باید دقیقاً مطابقت داشته باشند و اگر متد یک تابع باشد، مقدار برگشتی نیز باید مطابقت داشته باشد.

اعلان‌های متد باید حاوی راهنماهای ویژه‌ای باشند که با روال‌ها و توابع دیگر به کار برده نمی‌شوند. راهنماها تنها باید در اعلان کلاس ظاهر شوند، نه در تعریف اعلان، و بایستی همواره به ترتیب زیر لیست شده باشند:

```
reintroduce; overload; binding; calling convention; abstract; warning
```

جایی که *binding*، می‌تواند **virtual**، **dynamic** یا **override** باشد؛ *calling convention* می‌تواند **register**، **pascal**، **cdecl**، **stdcall** یا **safecall** باشد؛ و *warning* می‌تواند **platform**، **deprecated** یا **library** باشد.

Inherited

واژه کلیدی **inherited** نقش ویژه ای در پیاده‌سازی رفتار چندریخت[۱] ایفا می‌کند. **inherited** در تعاریف متد، با یا بدون یک شناسه بعد از آن، ظاهر می‌شود. اگر بعد از **inherited** اسم یک عضو بیاید، بیانگر یک فراخوان متد نرمال یا ارجاع به یک خاصیت یا فیلد خواهد بود — به استثنای این که جستجو برای عضو ارجاع شده با نیای بی‌واسطه کلاس متد محیطی، آغاز می‌شود. برای مثال، زمانی که

```
inherited Create(...);
```

در تعریف یک متد ظاهر می‌شود، *Create* موروثی را فرامی‌خواند.

هرگاه بعد از **inherited** هیچ شناسه‌ای وجود نداشته باشد، به متد موروثی با همان نام به عنوان متد محیطی اشاره می‌کند. در این حالت، **inherited** هیچ پارامتر صریحی نمی‌گیرد، اما به متد موروثی همان پارامترها ارسال می‌شوند که با آنها متد محیطی فراخوان شده بود. برای مثال،

```
inherited;
```

بارها در پیاده‌سازی سازنده‌ها رخ می‌دهد. این دستور سازنده موروثی (inherited) را با همان پارامترهایی که به فرزند ارسال شده بودند، فرامی‌خواند.

---

[۱] Polymorphic

Self

در میان پیاده‌سازی یک متد، شناسه *Self* به شیئی اشاره می‌کند که متد در آن فراخوان شده است. برای مثال، در زیر پیاده‌سازی متد *Add* برای *TCollection* که در یونیت *Classes* جای دارد، آورده شده است.

```
function TCollection.Add: TCollectionItem;
begin
Result := FItemClass.Create(Self);
end;
```

متد *Add*، متد *Create* را در کلاس اشاره شده توسط فیلد *FItemClass* فرامی‌خواند، که همواره یک فرزند *TCollectionItem* است. *TCollectionItem.Create* پارامتر واحدی از نوع *TCollection* می‌گیرد، از این رو جایی که *Add* فراخوان شده باشد، *Add* شیء وهله *TCollection* را به آن ارسال می‌کند. این امر در مثال زیر تشریح شده است.

```
var MyCollection: TCollection;
...
MyCollection.Add          // MyCollection is passed to the TCollectionItem.Create method
```

Self برای مقاصد گوناگونی سودمند است. برای مثال، یک شناسه عضو اعلان شده در نوع کلاس ممکن است در بلوک یکی از متدهای کلاس اعلان مجدد شده باشد. در این حالت، شما می‌توانید به شناسه عضو اصلی به صورت Self.*Identifier* دسترسی پیدا کنید. برای آگاهی از اطلاعات بیشتر درباره *Self* در متدهای کلاس، بخش «متدهای کلاسی» را در همین فصل ملاحظه نمایید.

انقیاد متد

متدها می‌توانند *استاتیک*[۱] (پیش فرض)، *مجازی*[۲] یا *پویا*[۳] باشند. متدهای پویا و مجازی می‌توانند *ابطال شده* و *از نو تعریف شوند*[۴]؛ هم چنین آنها می‌توانند مجرد[۵] باشند. این نقش‌ها زمانی وارد بازی می‌شوند

---

۱ Static

۲ Virtual

۳ Dynamic

۴ Overridden

۵ Abstract

که متغیری از یک نوع کلاس مقداری از یک نوع کلاس وارث (یا فرزند) را نگه دارد. زمانی که یک متد فراخوان شود آنها مشخص می‌کنند که کدام پیاده‌سازی فعال شود.

متدهای استاتیک

متدها به طور پیش فرض استاتیک هستند. هروقت که یک متد استاتیک فراخوان شود، نوع اعلان شده (زمان کامپایل) از متغیر کلاس یا شیء که در فراخوان متد به کار رفته، مشخص می‌کند که کدام پیاده‌سازی به کار بیفتد. در مثال زیر، متدهای *Draw* *استاتیک* هستند.

```
type
TFigure = class
procedure Draw;
end;
TRectangle = class(TFigure)
procedure Draw;
end;
```

با اعلانات داده شده در بالا، کد بعد نتیجه فراخوانی یک متد استاتیک را تشریح می‌کند. در فراخوان دوم به *Figure.Draw*، متغیر *Figure* به یک شیء از کلاس *TRectangle* ارجاع (یا اشاره) می‌کند، اما فراخوانی پیاده‌سازی *Draw* در *TFigure* را احضار می‌کند، زیرا نوع اعلان شده متغیر *Figure*، *TFigure* است.

```
var
Figure: TFigure;
Rectangle: TRectangle;
begin
Figure := TFigure.Create;
Figure.Draw;                    // calls TFigure.Draw
Figure.Destroy;
Figure := TRectangle.Create;
Figure.Draw;                    // calls TFigure.Draw
TRectangle(Figure).Draw;        // calls TRectangle.Draw
Figure.Destroy;
Rectangle := TRectangle.Create;
Rectangle.Draw;                  // calls TRectangle.Draw
Rectangle.Destroy;
end;
```

متدهای پویا و مجازی

برای ایجاد یک متد پویا یا مجازی، راهنمای **virtual** یا **dynamic** را ضمیمه اعلان کنید. متدهای مجازی و پویا، برخلاف متدهای استاتیک، می‌توانند در کلاس‌های فرزند باطل شده و از نو تعریف

شوند. هر وقت متدی که تعریف مجدد شده است، فراخوانی شود، نوع واقعی (زمان اجرا) کلاس یا شیء استفاده شده در فراخوان متد— نه نوع اعلان شده متغیر— تعیین می‌کند که کدام پیاده‌سازی فعال باشد.

برای تعریف مجدد یک متد، آن را با راهنمای **override** اعلان مجدد کنید. یک اعلان **override** باید با اعلان نیا در ترتیب و نوع پارامترهایش و در نوع نتیجه‌اش (اگر داشته باشد) منطبق باشد. در مثال زیر، متد *Draw* که در *TFigure* اعلان شده است، در دو کلاس فرزند باطل شده است.

```
type
TFigure = class
procedure Draw; virtual;
end;
TRectangle = class(TFigure)
procedure Draw; override;
end;
TEllipse = class(TFigure)
procedure Draw; override;
end;
```

با این اعلانات داده شده، کد زیر نتیجه فراخوان یک متد مجازی را از طریق متغیری که نوع واقعی‌اش در زمان اجرا تغییر می‌کند، تشریح می‌کند.

```
var
Figure: TFigure;
begin
Figure := TRectangle.Create;
Figure.Draw;                    // calls TRectangle.Draw
Figure.Destroy;
Figure := TEllipse.Create;
Figure.Draw;                    // calls TEllipse.Draw
Figure.Destroy;
end;
```

تنها متدهای پویا و مجازی می‌توانند باطل شده و از نو تعریف گردند. گرچه همه متدها می‌توانند سربارگذاری شوند. بخش «سربارگذاری متدها» را ملاحظه نمایید.

مجازی در برابر پویا

متدهای پویا و مجازی از نظر معناشناختی معادل هستند. آنها تنها در توزیع پیاده‌سازی فراخوان متد در زمان اجرا اختلاف دارند. متدهای مجازی برای سرعت بهینه سازی انجام می‌دهند، در حالی که متدهای پویا برای اندازه کد بهینه‌سازی می‌کنند.

به طور کلی، متدهای مجازی کارامدترین روش برای پیاده‌سازی رفتار چندریخت هستند. زمانی که یک کلاس پایه متدهای ابطال‌پذیر زیادی را اعلان می‌کند که در یک برنامه، توسط کلاس‌های فرزند بسیاری به ارث برده می‌شوند، اما تنها برخی اوقات باطل می‌شوند، متدهای پویا سودمند خواهند بود.

تعریف مجدد در برابر مخفی‌سازی

چنان چه اعلان یک متد، شناسه متد و امضای پارامتر یکسانی را به عنوان یک متد موروثی تعیین کند، اما **override** را ضمیمه نکند، اعلان جدید صرفاً متد موروثی را بدون باطل کردن و تعریف مجدد آن مخفی می‌کند. هر دو متد واقع در کلاس فرزند، جایی که نام متد به طور استاتیک محدود شده است، موجود هستند. برای مثال،

```
type
T1 = class(TObject)
procedure Act; virtual;
end;
T2 = class(T1)
procedure Act;                    // Act is redeclared, but not overridden
end;
var
SomeObject: T1;
begin
SomeObject := T2.Create;
SomeObject.Act;                   // calls T1.Act
end;
```

Reintroduce

راهنمای **reintroduce** مانع هشدارهای کامپایلر درباره مخفی‌سازی متدهای مجازی که پیش از این اعلان شده، می‌شود. برای مثال،

```
procedure DoSomething; reintroduce;     // the ancestor class also has a DoSomething method
```

چنان چه می‌خواهید یک متد مجازی موروثی را با یک متد تازه مخفی کنید از **reintroduce** استفاده کنید.

متدهای مجرد

یک متد مجرد (abstract) یک متد مجازی یا پویاست که هیچ پیاده‌سازی در کلاس جایی که اعلان شده، ندارد. پیاده‌سازی این متد تا یک کلاس فرزند به تعویق افتاده است. متدهای مجرد بایستی با راهنمای **abstract** بعد از **virtual** یا **dynamic** اعلان شده باشند. برای مثال،

```
procedure DoSomething; virtual; abstract;
```

شما می‌توانید یک متد مجرد را تنها در یک کلاس یا وهله‌ای از یک کلاس که متد در آن باطل گردیده و از نو تعریف شده است، فراخوانی کنید.

سربارگذاری متدها

یک متد می‌تواند با استفاده از راهنمای **overload** اعلان مجدد شود. در این حالت، اگر متد اعلان مجدد شده امضای پارامتر متفاوتی از نیای خود داشته باشد، متد موروثی را بدون مخفی‌سازی آن سربارگذاری می‌کند. فراخوانی متد در یک کلاس فرزند هر پیاده‌سازی را که پارامترهایش را با پارامترهای واقع در فراخوان مطابقت دهد، فعال می‌کند.

اگر یک متد مجازی را سربارگذاری می‌کنید، چنان چه آن را در کلاس‌های فرزند اعلان مجدد کردید، از راهنمای **reintroduce** استفاده کنید.

```
type
T1 = class(TObject)
procedure Test(I: Integer); overload; virtual;
end;
T2 = class(T1)
procedure Test(S: string); reintroduce; overload;
end;
...
SomeObject := T2.Create;
SomeObject.Test('Hello!');              // calls T2.Test
SomeObject.Test(7);                     // calls T1.Test
```

در میان یک کلاس، نمی‌توانید چندین متد سربارگذاری شده را با نام یکسان منتشر کنید. صیانت از اطلاعات نوع زمان اجرا نیازمند یک نام منحصر به فرد برای هر عضو منتشر شده (published) است.

```
type
TSomeClass = class
published
function Func(P: Integer): Integer;
function Func(P: Boolean): Integer      // error
...
```

متدهایی که همانند تصریح کننده‌های **read** یا **write** خاصیت به کار گرفته می‌شوند، نمی‌توانند سربارگذاری شوند. پیاده‌سازی یک متد سربارگذاری شده باید لیست پارامترها را از اعلان کلاس

تکرار کند. برای آگاهی از اطلاعات بیشتر درباره سربارگذاری، بخش «سربارگذاری توابع و روال‌ها» را در فصل۶ ملاحظه نمایید.

سازنده‌ها

سازنده، متد ویژه‌ای است که وهله اشیاء را ایجاد و مقداردهی اولیه می‌کند. اعلان یک سازنده شبیه اعلان یک روال است، اما با کلمه **constructor** آغاز می‌شود. مثال‌ها:

```
constructor Create;
constructor Create(AOwner: TComponent);
```

سازنده‌ها باید از قرارداد فراخوان پیش فرض **register** استفاده کنند. با این که اعلان هیچ مقدار برگشتی را تعیین نمی‌کند، یک سازنده یک ارجاع (یا اشاره) به شیئی که ایجاد می‌کند یا از درون آن خوانده می‌شود، برمی‌گرداند.

یک کلاس می‌تواند بیشتر از یک سازنده داشته باشد، اما بیشتر کلاس‌ها تنها یک سازنده دارند. فراخوانی سازنده *Create* یک امر قراردادی است. برای ایجاد یک شیء، متد سازنده واقع در یک نوع کلاس را فراخوان کنید. برای مثال،

```
MyObject := TMyClass.Create;
```

انباره لازم را برای شیء جدید در روی پشته تخصیص می‌دهد، مقادیر همه فیلدهای ترتیبی را به صفر می‌نشاند، **nil** را به همه فیلدهای اشاره‌گر و نوع کلاس تخصیص می‌دهد و تمامی فیلدهای رشته را خالی می‌کند. دیگر بخش‌های تعیین شده در پیاده‌سازی سازنده بعداً انجام می‌شوند؛ نوعاً اشیاء براساس مقادیر ارسال شده به عنوان پارامتر برای سازنده، مقداردهی اولیه می‌شوند. بالاخره سازنده یک ارجاع به شیئی که به تازگی تخصیص حافظه شده و مقداردهی اولیه شده است، برمی‌گرداند. نوع مقدار برگردانده شده همان نوع کلاس تعیین شده در فراخوان سازنده است. اگر در طول اجرای یک سازنده که در مورد یک ارجاع کلاس احضار شده بود، استثنایی بروز کند، تخریب کننده *Destroy* به طور خودکار برای تخریب شیء ناتمام فراخوان می‌شود.

زمانی که یک سازنده با استفاده از یک ارجاع شیء (به جای یک ارجاع کلاس) فراخوان شود، شیئی را ایجاد نمی‌کند. در عوض، سازنده، تنها با اجرای دستورات واقع در پیاده‌سازی سازنده، روی شیء تعیین شده عمل می‌کند، و سپس یک ارجاع (یا اشاره) به شیء را برمی‌گرداند. یک سازنده نوعاً روی

یک ارجاع شیء در پیوستگی با کلمه رزرو شده **inherited** برای اجرای یک سازنده موروثی، احضار می‌شود. در این جا مثالی از یک نوع کلاس و سازنده اش آورده شده است.

```
type
TShape = class(TGraphicControl)
private
FPen: TPen;
FBrush: TBrush;
procedure PenChanged(Sender: TObject);
procedure BrushChanged(Sender: TObject);
public
constructor Create(Owner: TComponent); override;
destructor Destroy; override;
...
end;

constructor TShape.Create(Owner: TComponent);
begin
inherited Create(Owner);              // Initialize inherited parts
Width := 65;                          // Change inherited properties
Height := 65;
FPen := TPen.Create;                  // Initialize new fields
FPen.OnChange := PenChanged;
FBrush := TBrush.Create;
FBrush.OnChange := BrushChanged;
end;
```

معمولاً اولین عمل یک سازنده فراخوانی یک سازنده موروثی به منظور مقداردهی اولیه فیلدهای موروثی شیء است. پس از آن سازنده فیلدهای معرفی شده در کلاس فرزند را مقداردهی اولیه می‌کند. از آن جا که یک سازنده همواره انباره حافظه‌ای را که برای یک شیء جدید تخصیص داده، پاک می‌کند، تمامی فیلدها با یک مقدار صفر (برای انواع ترتیبی)، **nil** (برای انوع اشاره‌گر و کلاس)، تهی (برای انواع رشته) یا *Unassigned* (برای واریانت‌ها) راه‌اندازی می‌شوند. از این رو دیگر نیازی به مقداردهی فیلدها در پیاده‌سازی یک سازنده نیست البته به استثنای مقادیر غیرصفر یا غیرتهی.

یک سازنده اعلان شده به صورت **virtual** چنان چه از طریق یک شناسه نوع کلاس احضار شود، با یک سازنده استاتیک معادل خواهد بود. گرچه هنگام ترکیب شدن با انواع ارجاع کلاس، سازنده‌های مجازی اجازه ساختار چندریخت اشیاء — یعنی ساختار اشیایی که نوع آنها در زمان کامپایل شناخته نمی‌شود— را می‌دهد. (بخش «ارجاعات کلاس» را در همین فصل ملاحظه نمایید.)

تخریب کننده‌ها

تخریب کننده، متد ویژه‌ای است که شیء را درجایی که فراخوانده شده، از بین برده و حافظه‌اش را آزاد می‌کند. اعلان یک تخریب کننده شبیه اعلان یک روال به نظر می‌رسد، اما با کلمه **destructor** آغاز می‌شود. چند نمونه:

```
destructor Destroy;
destructor Destroy; override;
```

تخریب کننده‌ها باید از قرارداد فراخوان پیش فرض **register** استفاده کنند. با این که یک کلاس می‌تواند بیشتر از یک تخریب کننده داشته باشد، اما توصیه می‌شود که هر کلاس متد *Destroy* موروثی را باطل کرده و آن را از نو تعریف نموده و هیچ تخریب کننده دیگری اعلان نکند. برای فراخوانی یک تخریب کننده، باید به یک شیء نمونه ارجاع (یا اشاره) کنید. برای مثال،

```
MyObject.Destroy;
```

هر زمان که یک تخریب کننده فراخوان شود، ابتدا اعمال تعیین شده در پیاده‌سازی تخریب کننده اجرا می‌شوند. نوعاً این عملیات‌ها شامل از بین بردن هر یک از اشیاء تعبیه شده و آزادسازی منابعی که توسط شیء اشغال شده بودند، می‌باشد. سپس انباره‌ای که برای شیء تخصیص داده شده بود آزاد می‌شود. در این جا مثالی از پیاده‌سازی یک تخریب کننده آورده شده است.

```
destructor TShape.Destroy;
begin
FBrush.Free;
FPen.Free;
inherited Destroy;
end;
```

آخرین عملی که در پیاده‌سازی یک تخریب کننده صورت می‌گیرد، نوعاً فراخوانی تخریب کننده موروثی برای از بین بردن فیلدهای موروثی شیء است.

چنان چه یک استثناء در طی ایجاد یک شیء صادر شود، *Destroy* به طور خودکار برای آزادسازی شیء ناتمام فراخوانده می‌شود. این حرف به معنای این است که *Destroy* بایستی برای آزادسازی اشیائی که به طور جزیی ساخته شده‌اند، آمادگی داشته باشد. از آن جایی که یک تخریب کننده قبل از انجام اعمال دیگر، فیلدهای یک شیء جدید را برابر صفر یا مقادیر تهی قرار می‌دهد، فیلدهای نوع کلاس و نوع اشاره‌گر در شیئی که تا اندازه‌ای ساخته شده (یعنی به طور جزیی)، همواره **nil** هستند. از این رو یک تخریب کننده قبل از عمل کردن بر روی فیلدهای نوع کلاس یا نوع اشاره‌گر، یک بررسی

برای مقادیر **nil** انجام دهد. فراخوانی متد *Free* (که در *TObject* تعریف شده است)، به جـای *Destroy*، یک روش سرراست و آسان را برای تحقیق درباره مقادیر **nil** قبل از تخریـب یـک شـیء بـه دسـت می‌دهد.

متدهای پیغام

متدهای پیغام[۱] عکسل‌العمل‌هایی را در مقابل پیغام‌هایی که به طور پویا ارسال می‌شوند، پیاده‌سازی می‌کنند. ترکیب نوشتاری/نحوی متد پیغام روی همه پلت‌فرم‌ها پشتیبانی می‌شود. VCL از متدهای پیغام برای پاسخ دادن به پیغام‌های ویندوز استفاده می‌کند. CLX برای پاسخ دادن به رویدادهای سیستم، از متدهای پیغام استفاده نمی‌کند.

یک متد پیغام با جای دادن راهنمای **message** در اعلان یک متد ایجاد می‌شود، کـه بعـد از آن یـک عدد صحیح ثابت در فاصله ۱ و ۴۹۱۵۱ می‌آید که ID پیغام را تعیین می‌کند. بـرای متـدهای پیغـام در کنترلهای VCL، عدد صحیح ثابت می‌تواند یکی از IDهای پیغام تعریف شده ویندوز باشد، همراه با انواع رکورد متناظر، در یونیت *Messages*. متد پیغام بایستی روالی باشد که پـارامتر **var** واحـدی مـی‌گیـرد. برای مثال، در ویندوز:

```
type
TTextBox = class(TCustomControl)
private
procedure WMChar(var Message: TWMChar); message WM_CHAR;
...
end;
```

برای مثال، در لینوکس یا برای برنامه‌نویسی پایگاه متقاطع، شما بایستی پیغام‌ها را به صورت زیـر اداره کنید:

```
const
ID_REFRESH = $0001;
type
TTextBox = class(TCustomControl)
private
procedure Refresh(var Message: TMessageRecordType); message ID_REFRESH;
...
end;
```

---

[۱] Message methods

یک متد پیغام نبایستی دارای راهنمای **override** برای تعریف مجدد یک متد پیغام موروثی باشد. در واقع، متد پیغام نباید نام متد یا نوع پارامتر یکسانی را به صورت متدی که باطل کرده و از نوع تعریف می‌کند، تعیین کند. ID پیغام تنها مشخص می‌کند که متد به کدام پیغام پاسخ دهد و این که آیا آن یک override است یا نه.

پیاده‌سازی متدهای پیغام

پیاده‌سازی یک متد پیغام می‌تواند متد پیغام موروثی را فراخوانی کند، مانند مثال زیر (برای ویندوز):

```
procedure TTextBox.WMChar(var Message: TWMChar);
begin
if Chr(Message.CharCode) = #13 then
ProcessEnter
else
inherited;
end;
```

در لینوکس یا برنامه‌نویسی پایگاه متقاطع، همان مثال را می‌توانید به صورت زیر بنویسید:

```
procedure TTextBox.Refresh(var Message: TMessageRecordType);
begin
if Chr(Message.Code) = #13 then
...
else
inherited;
end;
```

دستور **inherited** با برگشت به عقب و جستجو از طریق سلسله مراتب کلاس، اولین متد پیغام با همان ID را به عنوان متد جاری احضار می‌کند، و به طور خودکار رکورد پیغام را به آن ارسال می‌کند. اگر هیچ کلاس نیایی یک متد پیغام برای ID داده شده را اجرا نکند، **inherited** متد *DefaultHandler* را که در اصل در *TObject* تعریف شده است، فرامی‌خواند.

پیاده‌سازی *DefaultHandler* در *TObject* در اصل بدون اجرای هر عملی برگردانده می‌شود. با ابطال و تعریف مجدد *DefaultHandler*، یک کلاس می‌تواند مدیریت پیغام‌های پیش فرض خود را پیاده‌سازی کند. در ویندوز، متد *DefaultHandler* برای کنترل‌های VCL، تابع *DefWindowProc* را فرامی‌خواند.

توزیع پیغام

گرداننده‌های پیغام به ندرت به طور مستقیم فراخوانده می‌شوند. در عوض، پیغام‌ها با استفاده از متد *Dispatch* که از *TObject* به ارث رسیده است، به یک شیء توزیع می‌شوند:

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

پارامتر *Message* ارسال شده به *Dispatch* باید یک رکورد باشد که اولین مدخلش فیلدی از نوع *Cardinal* حاوی یک ID پیغام است.

Dispatch با برگشت به عقب و جستجو از طریق سلسله مراتب کلاس (با شروع از کلاس شیء جایی که فراخوانده شده) اولین متد پیغام برای ID ارسال شده به خود را، احضار می‌کند. چنان چه هیچ متد پیغامی برای ID داده شده یافت نشود، *Dispatch* متد *DefaultHandler* را فرامی‌خواند.

خاصیت‌ها

یک خاصیت، مانند یک فیلد، مشخصه‌ای از یک شیء را تعریف می‌کند. اما در حالی که یک فیلد صرفاً یک موقعیت انباره است که محتویاتش می‌تواند بازرسی شده و تغییر کند، یک خاصیت اعمال مخصوصی را با خواندن یا جرح وتعدیل داده‌اش همبسته ساخته و پیوند می‌دهد. خاصیت‌ها کنترلی برای دسترسی به مشخصه‌های یک شیء فراهم می‌کنند و به مشخصه‌ها اجازه می‌دهند تا محاسبه شوند. اعلان یک خاصیت، یک نام و یک نوع تعیین کرده و حداقل یک تصریح کننده دسترسی را دربردارد. ترکیب نوشتاری/نحوی اعلان یک خاصیت به صورت زیر است

```
property propertyName[indexes]: type index integerConstant specifiers;
```

جایی که

- *propertyName* هرگونه شناسه معتبری است.
- [*indexes*] اختیاری بوده و دنباله‌ای از اعلان پارامترامترهاست که توسط نقطه ویرگول‌ها از هم جدا می‌شوند. هر اعلان پارامتر قالب *identifier*1, ..., *identifier*n: *type* را دارد. برای آگاهی از اطلاعات بیشتر، بخش «خاصیت‌های آرایه» را در همین فصل ببینید.
- *type* بایستی یک نوع از پیش تعریف شده یا نوعی داده از قبل اعلان شده باشد. یعنی، اعلان‌های خاصیتی مانند **property** Num: 0..9 ... معتبر هستند.
- شرط *integerConstant* index اختیاریست. برای اطلاعات بیشتر، بخش «تصریح کننده‌های اندیس» را در همین فصل ملاحظه نمایید.
- *specifiers* دنباله‌ای از تصریح کننده‌های **read**، **write**، **stored**، **default** (یا **nodefault**) و **implements** است. هر اعلان خاصیت باید دست کم یکی از تصریح کننده‌های **read** یا

write را دارا باشد. (برای اطلاعات بیشتر درباره **implements**، بخش «پیاده‌سازی واسط‌ها توسط نمایندگی» را در فصل ۱۰ ملاحظه نمایید.)

خاصیت‌ها توسط تصریح کننده‌های دسترسی‌شان تعریف می‌شوند. برخلاف فیلدها، خاصیت‌ها نمی‌توانند به صورت پارامترهای **var** ارسال شوند، یا عملگر **@** هم نمی‌تواند به یک خاصیت اعمال شود. دلیلش این است که یک خاصیت لزوماً در حافظه وجود ندارد. برای مثال، خاصیت می‌تواند یک متد **read** داشته باشد که مقداری را از یک پایگاه داده بازیابی کند یا یک مقدار تصادفی تولید کند.

دسترسی خاصیت

هر خاصیت یک تصریح کننده **read**، یک تصریح کننده **write**، یا هردو را داراست. اینها تصریح کننده‌های دسترسی خوانده شده و قالب زیر را دارند

```
read fieldOrMethod
write fieldOrMethod
```

جایی که *fieldOrMethod* اسم یک فیلد یا متد اعلان شده در همان کلاس به عنوان خاصیت یا اعلان شده در یک کلاس نیا[۱]ست.

- اگر *fieldOrMethod* در همان کلاس اعلان شده باشد، باید قبل از اعلان خاصیت رخ دهد. اگر در یک کلاس نیا اعلان شده باشد، باید از کلاس فرزند قابل مشاهده باشد؛ یعنی، نمی‌تواند یک فیلد یا متد خصوصی (private) متعلق به یک کلاس نیای اعلان شده در یک یونیت متفاوت باشد.
- اگر *fieldOrMethod* یک فیلد باشد، باید از همان نوعی باشد که خاصیت آن نوع را دارد.
- اگر *fieldOrMethod* یک متد باشد، نمی‌تواند سربارگذاری شده باشد. علاوه بر این، متدهای دسترسی برای یک خاصیت منتشرشده (published) باید از قرارداد فراخوانی پیش فرض **register** استفاده کنند.
- در یک تصریح کننده **read**، اگر *fieldOrMethod* یک متد باشد، این متد باید تابع بدون پارامتری باشد که نوع نتیجه برگشتی‌اش همان نوع خاصیت است.

---

[۱] Ancestor class

- در یک تصریح کننده **write**، چنان چه *fieldOrMethod* یک متد باشد، این متد باید روالی باشد که مقداری واحد یا پارامتری **const** از همان نوع خاصیت می‌گیرد.

برای مثال، با اعلان‌های داده شده زیر

```
property Color: TColor read GetColor write SetColor;
```

متد *GetColor* باید به صورت زیر اعلان شود

```
function GetColor: TColor;
```

و متد *SetColor* باید مانند یکی از این حالات اعلان شود:

```
procedure SetColor(Value: TColor);
procedure SetColor(const Value: TColor);
```

(البته اسم پارامتر *SetColor* نبایستی *Value* باشد.)

زمانی که یک خاصیت در یک عبارت مورد ارجاع باشد، مقدارش با استفاده از فیلد یا متد لیست شده در تصریح کننده **read** خوانده می‌شود. هرگاه در یک دستور تخصیص به یک خاصیت ارجاع شود، مقدارش با استفاده از فیلد یا متد فهرست شده در تصریح کننده **write** نوشته می‌شود.

مثال زیر یک کلاس به نام *TCompass* را با یک خاصیت منتشر شده (published) به نام *Heading* اعلان می‌کند. مقدار *Heading* از طریق فیلد *FHeading* خوانده شده و از طریق روال *SetHeading* نوشته می‌شود.

```
type
THeading = 0..359;
TCompass = class(TControl)
private
FHeading: THeading;
procedure SetHeading(Value: THeading);
published
property Heading: THeading read FHeading write SetHeading;
...
end;
```

با این اعلان‌های داده شده، دستورات

```
if Compass.Heading = 180 then GoingSouth;
Compass.Heading := 135;
```

معادل هستند با

```
if Compass.FHeading = 180 then GoingSouth;
Compass.SetHeading(135);
```

در کلاس *TCompass*، هیچ عملیاتی با خواندن خاصیت *Heading*، مرتبط نشده است؛ عملیات **read** از بازیابی مقدار ذخیره شده در فیلد *FHeading* تشکیل شده است. به عبارت دیگر، تخصیص یک مقدار به خاصیت *Heading* به یک فراخوان به متد *SetHeading* برگردانده می‌شود، که احتمالاً، مقدار جدید را در فیلد *FHeading* ذخیره می‌کند به علاوه انجام اعمال دیگر. برای مثال، *SetHeading* ممکن است به این صورت پیاده‌سازی شود:

```
procedure TCompass.SetHeading(Value: THeading);
begin
if FHeading <> Value then
begin
FHeading := Value;
Repaint; // update user interface to reflect new value
end;
end;
```

یک خاصیت که اعلانش تنها شامل یک تصریح کننده **read** باشد، یک خاصیت فقط-خواندنی (*read-only*) خواهد بود و خاصیتی که اعلانش تنها شامل یک تصریح کننده **write** باشد یک خاصیت فقط-نوشتنی (*write-only*) خواهد بود. تخصیص یک مقدار به یک خاصیت فقط-خواندنی یا استفاده از یک خاصیت فقط-نوشتنی در یک عبارت، خطا محسوب می‌شود.

خاصیت‌های آرایه

خاصیت‌های آرایه[۱] خاصیت‌های اندیس‌دار شده هستند. آنها می‌توانند اشیاء را مانند اقلام واقع در یک لیست، کنترلهای فرزند یک کنترل و پیکسل‌های یک بیت مپ[۲] بیان کنند.

اعلان یک خاصیت آرایه‌ای یک لیست پارامتر دارد که اسامی و نوع زیرنویس‌ها را تعیین می‌کند. برای مثال،

```
property Objects[Index: Integer]: TObject read GetObject write SetObject;
property Pixels[X, Y: Integer]: TColor read GetPixel write SetPixel;
property Values[const Name: string]: string read GetValue write SetValue;
```

فرمت یک لیست پارامتر اندیس‌دار شده مشابه یک لیست پارامتر تابع یا روال است، به جز این که اعلان‌های پارامتر به جای پارانتزها در میان کروشه‌ها—**[ ]**— محصور می‌شوند. برخلاف آرایه‌ها، که

---

۱ Array properties

۲ Bitmap

تنها می‌توانند از زیرنویس‌های نوع ترتیبی استفاده کنند، خاصیت‌های آرایه‌ای اجازه زیرنویس‌هایی از هر نوع را می‌دهند.

برای خاصیت‌های آرایه‌ای، تصریح کننده‌های دسترسی باید به جای فیلدها متدها را لیست کنند. در یک تصریح کننده **read**، متد باید تابعی باشد که تعداد و نوع پارامترهای لیست شده در لیست پارامتر اندیس‌دار خاصیت را، درست به همان ترتیب می‌گیرد و نوع نتیجه برگشتی‌اش برابر با نوع خاصیت است. در یک تصریح کننده **write**، متد باید روالی باشد که تعداد و نوع پارامترهای لیست شده در لیست پارامتر اندیس‌دار خاصیت را، درست به همان ترتیب به اضافه یک مقدار اضافی یا پارامتر **const** از همان نوع خاصیت، می‌گیرد. برای مثال، متدهای دسترسی برای خاصیت‌های آرایه واقع در بالا می‌توانند به این صورت اعلان شوند

```
function GetObject(Index: Integer): TObject;
function GetPixel(X, Y: Integer): TColor;
function GetValue(const Name: string): string;
procedure SetObject(Index: Integer; Value: TObject);
procedure SetPixel(X, Y: Integer; Value: TColor);
procedure SetValue(const Name, Value: string);
```

یک خاصیت آرایه‌ای توسط اندیس‌دار کردن شناسه خاصیت قابل دسترسی است. برای مثال، دستورات

```
if Collection.Objects[0] = nil then Exit;
Canvas.Pixels[10, 20] := clRed;
Params.Values['PATH'] := 'C:\DELPHI\BIN';
```

متناظر است با

```
if Collection.GetObject(0) = nil then Exit;
Canvas.SetPixel(10, 20, clRed);
Params.SetValue('PATH', 'C:\DELPHI\BIN');
```

در لینوکس، به جای 'C:\DELPHI\BIN' در مثال بالایی، می‌توانید از یک مسیر مانند '/usr/local/bin' استفاده کنید.

تعریف یک خاصیت آرایه‌ای می‌تواند با راهنمای **default** پی گرفته شود، که در این حالت خاصیت آرایه‌ای، خاصیت پیش فرض برای کلاس می‌شود. برای مثال،

```
type
TStringArray = class
public
property Strings[Index: Integer]: string ...; default;
...
end;
```

چنان چه کلاسی یک خاصیت پیش فرض داشته باشد، شما می‌توانید با اختصار نویسی *object*[*index*] به آن خاصیت دسترسی پیدا کنید، که این اختصارنویسی معادل با *object.property*[*index*] است. برای مثال، با اعلان‌های داده شده پیشین، StringArray.Strings[7] می‌تواند به صورت StringArray[7] کوتاه شده باشد. یک کلاس تنها می‌تواند یک خاصیت پیش فرض داشته باشد. تغییر یا پنهان‌سازی خاصیت پیش فرض در کلاس‌های فرزند ممکن است منجر به رفتار غیرمنتظره‌ای شود، زیرا کامپایلر همواره به طور استاتیک خاصیت پیش فرض یک شیء را تعیین می‌کند.

تصریح کننده‌های اندیس

تصریح کننده‌های اندیس به چندین خاصیت اجازه می‌دهند تا متد دسترسی یکسانی را به اشتراک گذارند در حالی که هر کدام بیانگر مقادیر متفاوتی هستند. تصریح کننده اندیس از راهنمای **index** که پس از آن یک ثابت عددی صحیح در فاصله 2147483647– و 2147483647 می‌آید، تشکیل می‌شود. چنان چه خاصیتی یک تصریح کننده اندیس داشته باشد، تصریح کننده‌های **read** و **write** آن باید به جای فیلدها، متدها را لیست کنند. برای مثال،

```
type
TRectangle = class
private
FCoordinates: array[0..3] of Longint;
function GetCoordinate(Index: Integer): Longint;
procedure SetCoordinate(Index: Integer; Value: Longint);
public
property Left: Longint index 0 read GetCoordinate write SetCoordinate;
property Top: Longint index 1 read GetCoordinate write SetCoordinate;
property Right: Longint index 2 read GetCoordinate write SetCoordinate;
property Bottom: Longint index 3 read GetCoordinate write SetCoordinate;
property Coordinates[Index: Integer]: Longint read GetCoordinate write SetCoordinate;
...
end;
```

یک متد دسترسی برای یک خاصیت دارای یک تصریح کننده اندیس باید یک پارامتر مقدار اضافی از نوع *Integer* بگیرد. برای یک تابع **read**، این پارامتر بایستی آخرین پارامتر باشد؛ برای یک روال **write**، باید پارامتر دومی به آخر (مقدم بر پارامتری که مقدار خاصیت را تعیین می‌کند) باشد. زمانی که یک برنامه به خاصیت دسترسی پیدا می‌کند، ثابت عددی صحیح خاصیت به طور خودکار به متد دسترسی ارسال می‌شود. با اعلان‌های داده شده در بالا، چنان چه *Rectangle* از نوع *TRectangle* باشد، در این صورت

```
Rectangle.Right := Rectangle.Left + 100;
```

متناظر است با

```
Rectangle.SetCoordinate(2, Rectangle.GetCoordinate(0) + 100);
```

تصریح کننده‌های انباره

راهنماهای اختیاری **stored**، **default** و **nodefault** تصریح کننده‌های انباره خوانده می‌شوند. آنها تأثیری بر روی رفتار برنامه ندارند، اما شیوه‌ای را که اطلاعات نوع زمان اجرا (RTTI) نگه داشته می‌شوند، کنترل می‌کنند. به ویژه، تصریح کننده‌های انباره تعیین می‌کنند که آیا مقادیر خاصیت‌های منتشرشده (published) در فایل‌ها فرم ذخیره شوند یا نه.

راهنمای **stored** باید با *True*، *False*، اسم یک فیلد *Boolean* یا اسم یک متد بدون پارامتر که یک مقدار *Boolean* را برمی‌گرداند، پی گرفته شود. برای مثال،

```
property Name: TComponentName read FName write SetName stored False;
```

اگر یک خاصیت هیچ راهنمای **stored** نداشته باشد، این خاصیت به گونه‌ای رفتار می‌کند که انگار stored True تصریح شده است. راهنمای **default** باید با یک ثابت از همان نوعی که خاصیت دارد، پی گرفته شود. برای مثال،

```
property Tag: Longint read FTag write FTag default 0;
```

برای ابطال یک مقدار **default** موروثی بدون تعیین یک مقدار جدید، از راهنمای **nodefault** استفاده کنید. راهنماهای **default** و **nodefault** تنها برای انواع ترتیبی و برای انواع مجموعه، به شرط آن که حدود بالا و پایین نوع مبنای مجموعه مقادیر ترتیبی بین صفر و ۳۱ داشته باشند، پشتیبانی می‌شوند؛ اگر یک چنین خاصیتی بدون **default** یا **nodefault** اعلان شود، طوری رفتار می‌کند که انگار **nodefault** تصریح شده است. برای اعداد حقیقی، اشاره‌گرها و رشته‌ها، به ترتیب یک مقدار ضمنی **default** صفر، **nil** و " (یعنی رشته تهی) موجود است.

هنگام ذخیره‌سازی حالت یک جزء[۱]، تصریح کننده‌های انباره خاصیت‌های منتشرشده (published) جزء چک می‌شوند. اگر مقدار جاری یک خاصیت متفاوت از مقدار **default**اش باشد (یا اگر هیچ

[۱] Component

مقدار **default** موجود نباشد) و تصریح کننده **stored**، *True* باشد، در این صورت مقدار خاصیت ذخیره می‌شود. در غیر این صورت، مقدار خاصیت ذخیره نمی‌شود.

توجه تصریح کننده‌های انباره برای خاصیت‌های آرایه‌ای پشتیبانی نمی‌شوند. راهنمای **default** هنگامی که در اعلان یک خاصیت آرایه‌ای به کار برده می‌شود، معنای متفاوتی دارد. بخش «خاصیت‌های آرایه» را در همین فصل ملاحظه نمایید.

ابطال خاصیت‌ها و اعلان‌های مجدد

اعلان یک خاصیت که نوعی را مشخص نمی‌کند یک *ابطال خاصیت*[۱] خوانده می‌شود. ابطال خاصیت‌ها به شما اجازه می‌دهند که تصریح کننده‌ها یا سطح دسترسی موروثی یک خاصیت را تغییر دهید. ساده‌ترین ابطال صرفاً از واژه کلیدی **property** که با یک شناسه خاصیت موروثی پی گرفته می‌شود، تشکیل می‌شود؛ این قالب برای تغییر سطح دسترسی (یا میدان دید) یک خاصیت به کار برده می‌شود. برای مثال، چنان چه یک کلاس نیا یک خاصیت حفاظت شده (protected) را اعلان کند، یک کلاس مشتق شده می‌تواند آن خاصیت را در یک بخش عمومی(public) یا منتشرشده (published) کلاس اعلان مجدد کند. ابطال‌های خاصیت می‌توانند حاوی راهنماهای **read**، **write**، **stored**، **default** و **nodefault** باشند؛ هر یک از این قبیل راهنماها دستور موروثی متناظر را باطل می‌کنند. یک ابطال می‌تواند یک تصریح کننده دسترسی را جایگزین کند، یک تصریح کننده غایب را اضافه کند، یا سطح دسترسی یک خاصیت را افزایش دهد، اما نمی‌تواند یک تصریح کننده دسترسی را حذف کند یا سطح دسترسی یک خاصیت را کاهش دهد . یک ابطال می‌تواند حاوی راهنمای **implements** باشد، که به فهرست واسط‌های پیاده‌سازی شده بدون حذف یکی از موروثی‌ها، اضافه می‌شود. اعلان‌های زیر استفاده از ابطال خاصیت‌ها را تشریح می‌کنند.

```
type
TAncestor = class
...
protected
property Size: Integer read FSize;
property Text: string read GetText write SetText;
property Color: TColor read FColor write SetColor stored False;
...
end;
type
```

---

[۱] Property override

```
TDerived = class(TAncestor)
...
protected
property Size write SetSize;
published
property Text;
property Color stored True default clBlue;
...
end;
```

ابطال *Size* یک تصریح کننده **write** اضافه می‌کند تا اجازه دهد که خاصیت تغییر داده شود. ابطال‌های *Text* و *Color* سطح دسترسی خاصیت‌ها را از حفاظت شده (protected) به منتشر شده (published) تغییر می‌دهند. در ضمن ابطال خاصیت *Color* نیز مشخص می‌کند که چنان چه مقدار خاصیت *clBlue* نباشد، خاصیت بایستی فیلد باشد.

یک اعلان مربوط به یک خاصیت که حاوی یک شناسه نوع است خاصیت موروثی را به جای ابطال کردن، پنهان می‌کند. این سخن به معنای این است که یک خاصیت جدید با همان نام خاصیت موروثی، ایجاد می‌شود. هر اعلان خاصیت که یک نوع را تعیین می‌کند باید یک اعلان کامل باشد و از این رو بایستی دست کم دارای یکی از تصریح کننده‌های دسترسی باشد.

در یک کلاس مشتق شده، یک خاصیت خواه مخفی شده باشد یا خواه باطل شده باشد، جستجوی خاصیت همواره استاتیک خواهد بود. یعنی نوع اعلان شده (زمان کامپایل) متغیر به کار رفته برای شناسایی یک شیء، تعبیر شناسه‌های خاصیت آن را مشخص می‌کند. از این رو، بعد از اجرا شدن کد زیر، خواندن یا تخصیص یک مقدار به *MyObject.Value* یا *Method1* یا *Method2* را احضار می‌کند ولواین‌که *MyObject* یک وهله از *TDescendant* را نگه داشته باشد. اما شما می‌توانید *MyObject* را به *TDescendant* قالب‌بندی (تبدیل نوع صریح) کنید تا به خاصیت‌های کلاس فرزند و تصریح کننده‌های دسترسی آنها دسترسی پیدا کنید.

```
type
TAncestor = class
...
property Value: Integer read Method1 write Method2;
end;
TDescendant = class(TAncestor)
...
property Value: Integer read Method3 write Method4;
end;
var MyObject: TAncestor;
...
MyObject := TDescendant.Create;
```

ارجاعات کلاس

برخی اوقات عملیات‌ها به جای اینکه روی وهله‌های یک کلاس (یعنی اشیاء) انجام شوند، روی خود کلاس انجام می‌شوند. برای مثال، زمانی که یک متد سازنده را با استفاده یک ارجاع کلاس فرامی‌خوانید، این امر اتفاق می‌افتد. شما به یک کلاس معین همواره با استفاده از نامش ارجاع (یا اشاره) می‌کنید، اما گاهی لازم می‌شود تا متغیرها یا پارامترهایی که کلاس‌ها به عنوان مقادیر می‌گیرند، اعلان شوند و در این موقعیت‌ها شما نیازمند انواع *class-reference* خواهید بود.

انواع Class-reference

یک نوع ارجاع کلاس، که گاهی اوقات یک متا کلاس[۱] نیز خوانده می‌شود، توسط ساختاری به فرم زیر مشخص می‌شود

```
class of type
```

جایی که *type* هر نوع کلاسی بوده و شناسه *type* خودش مقداری را مشخص می‌کند که نوعش کلاسی از *type* است. اگر *type*1 یک نیای *type*2 باشد، در این صورت کلاسی از *type*2 سازگار برای تخصیص با کلاسی از نوع *type*1 است. از این رو

```
type TClass = class of TObject;
var AnyObj: TClass;
```

متغیری به نام *AnyObj* اعلان می‌کند که می‌تواند یک ارجاع به هر کلاسی را نگه دارد. (تعریف یک نوع class-reference نمی‌تواند به طور مستقیم در اعلان یک متغیر یا لیست پارامتر رخ دهد.) شما می‌توانید مقدار **nil** را به متغیری از هر نوع class-reference تخصیص دهید.

---

[۱] Metaclass

برای این که ببینید انواع class-reference چگونه به کـار بـرده مـی‌شـوند، اعـلان سـازنده مربـوط بـه *TCollection* (در یونیت *Classes*) را ملاحظه نمایید:

```
type TCollectionItemClass = class of TCollectionItem;
...
constructor Create(ItemClass: TCollectionItemClass);
```

این اعلان می‌گوید که برای ایجاد یک نمونه شیئی از *TCollection*، شما بایستی اسم یک کلاس در حال اشتقاق از *TCollectionItem* را به سازنده ارسال کنید.

هنگامی که می‌خواهید یک متد کلاسی یا سازنده مجازی را روی یک کلاس یا شیء — که مقدار واقعی‌اش در زمان کامپایل ناشناخته است— احضار کنید، انواع class-reference سودمند خواهند بود.

سازنده‌ها و ارجاعات کلاس

یک سازنده می‌تواند با استفاده از متغیری از یک نوع class-reference فراخوانده شود. این امر اجازه ساخت اشیائی را که نوعشان در زمان کامپایل ناشناخته است، می‌دهد. برای مثال،

```
type TControlClass = class of TControl;
function CreateControl(ControlClass: TControlClass;
const ControlName: string; X, Y, W, H: Integer): TControl;
begin
Result := ControlClass.Create(MainForm);
with Result do
begin
Parent := MainForm;
Name := ControlName;
SetBounds(X, Y, W, H);
Visible := True;
end;
end;
```

تابع *CreateControl* یک پارامتر class-reference لازم دارد تا به او بگوید که چه نوعی از کنترل ایجاد شود. تابع از این پارامتر استفاده می‌کند تا سازنده کلاس را فراخوانی کند. از آن جایی که شناسـه‌هـای نوع کلاس مقادیر class-reference را مشخص می‌کنند، یک فراخوان به *CreateControl* می‌تواند شناسه کلاس را برای ایجاد وهله‌ای از آن تعیین کند. برای مثال،

```
CreateControl(TEdit, 'Edit1', 10, 10, 100, 20);
```

سازنده‌های فراخوانده شده با استفاده از class referenceها معمولاً مجازی هستند. پیاده‌سازی سازنده فعال شده توسط فراخوانی به نوع زمان اجرای class reference بستگی دارد.

عملگرهای کلاس

هر کلاس متدهایی به نام *ClassType* و *ClassParent* را از *TObject* ارث می‌برد که به ترتیب، یک ارجاع به کلاس یک شیء و کلاس نیای بی‌واسطه یک شیء برمی‌گرداند. هر دو متد یک مقدار نوع *TClass* (جایی که TClass = class of TObject) را برمی‌گردانند، که می‌تواند به یک نوع ویژه‌تری قالب‌بندی شود. در ضمن هر کلاس یک متد به نام *InheritsFrom* را ارث می‌برد که این متد بررسی می‌کند که آیا شیء جایی که فراخوان شده، از یک کلاس مشخص شده‌ای مشتق می‌شود یا نه. این متدها توسط عملگرهای **is** و **as** به کار برده می‌شوند و به ندرت لازم می‌شود که آنها را به طور مستقیم فراخوانی کرد.

عملگر is

عملگر **is**، که بررسی نوع پویا را انجام می‌دهد، برای وارسی کلاس واقعی زمان اجرای یک شیء به کار برده می‌شود. عبارت

```
object is class
```

اگر *object* وهله‌ای از کلاس مشخص شده توسط *class* یا یکی از فرزندان آن باشد، *True* را برمی‌گرداند و درغیراین صورت *False* را برمی‌گرداند. (اگر *object* برابر **nil** باشد، نتیجه *False* خواهد بود.) اگر نوع اعلان شده *object* با *class* نامرتبط باشد— یعنی اگر انواع مجزایی باشند و یکی از آنها نیای دیگری نباشد— یک خطای کامپایل بروز می‌کند. برای مثال،

```
if ActiveControl is TEdit then TEdit(ActiveControl).SelectAll;
```

این دستور بعد از اولین بررسی که نشان دهد شیئی که بدان اشاره می‌کند وهله‌ای از *TEdit* یا یکی از فرزندانش است، یک متغیر را به *TEdit* قالب‌بندی می‌کند (یعنی تبدیل نوع صریح می‌کند).

عملگر as

عملگر **as** قالب‌بندی‌های بررسی شده را انجام می‌دهد. عبارت

```
object as class
```

یک ارجاع به شیئی یکسان با *object* برمی‌گرداند، اما با نوع داده شده توسط *class*. در زمان اجرا، *object* باید وهله‌ای از کلاس مشخص شده به وسیله *class* یا یکی از فرزندان آن باشد یا **nil** باشد؛

درغیراین صورت یک استثناء صادر می‌شود. اگر نوع اعلان شده *object* با *class* نامرتبط باشد— یعنی اگر از نوع‌های مجزایی باشند و یکی نیای دیگری نباشد— یک خطای کامپایل نتیجه می‌شود. برای مثال،

```
with Sender as TButton do
begin
Caption := '&Ok';
OnClick := OkClick;
end;
```

قواعد حق تقدم عملگر اغلب نیازمند این هستند که قالب‌بندی‌های **as** در میان پارانتزها محصور باشند. برای مثال،

```
(Sender as TButton).Caption := '&Ok';
```

متدهای کلاسی

یک متد کلاسی متدی (غیر از یک سازنده) است که به جای اشیاء روی کلاس‌ها عمل می‌کند. تعریف یک متد کلاسی باید با واژه کلیدی **class** آغاز شود. برای مثال،

```
type
TFigure = class
public
class function Supports(Operation: string): Boolean; virtual;
class procedure GetInfo(var Info: TFigureInfo); virtual;
...
end;
```

در ضمن تعریف اعلان یک متد کلاس هم بایستی با **class** آغاز شود. برای مثال،

```
class procedure TFigure.GetInfo(var Info: TFigureInfo);
begin
...
end;
```

در تعریف اعلان یک متد کلاس، شناسه *Self* کلاسی را بیان می‌کند که متد در آن فراخوان می‌شود (که می‌تواند فرزند کلاسی که در آن تعریف شده است باشد). اگر متد در کلاس C فراخوانده شود، در این صورت *Self* از نوع کلاس C خواهد بود. ازاین‌رو شما نمی‌توانید از *Self* برای دسترسی به فیلدها، خواص و متدهای (شیء) نرمال استفاده کنید، اما می‌توانید از آن برای فراخوان سازنده‌ها و متدهای کلاس دیگر استفاده کنید. یک متد کلاسی می‌تواند از طریق یک ارجاع کلاس یا یک ارجاع شیء فراخوانده شود. زمانی که از طریق یک ارجاع شیء فراخوانده شود، کلاس شیء مقدار *Self* می‌شود.

استثناها

یک **استثناء**[۱] زمانی که یک خطا یا رویداد دیگری در اجرای طبیعی یک برنامه وقفه می‌اندازد، صادر می‌شود. استثناء کنترل اجرا را به یک **گرداننده استثناء**[۲] منتقل می‌کند، که به شما اجازه می‌دهد تا منطق نرمال برنامه را از مدیریت استثناء جدا کنید. از آن جایی که استثناها اشیاء هستند، آنها می‌توانند با استفاده از توارث[۳] در سلسه مراتب‌هایی گروه‌بندی شده و استثناهای جدید می‌توانند بدون اثرگذاری بر کد جدید معرفی شوند. یک استثناء می‌تواند اطلاعاتی مانند یک پیغام خطا را از شیء، جایی که صادر شده به نقطه‌ای دیگر جایی که اداره می‌شود، حمل کند.

هرگاه یک برنامه از یونیت *SysUtils* استفاده کند، تمامی خطاهای حین اجرا به طور خودکار به استثناء تبدیل می‌شوند. خطاها می‌توانند گرفته شده و اداره شوند که در غیر این صورت به یک برنامه خاتمه می‌دهند— خطاهایی مانند حافظه ناکافی، تقسیم بر صفر و نیز خطاهای حفاظت عمومی.

چه زمانی از استثناها استفاده کنیم؟

استثناها روش ظریفی برای به دام انداختن خطاهای زمان اجرا به دست می‌دهند بدون این که برنامه متوقف شده و از دستورات شرطی غیر حرفه‌ای و نازیبا استفاده شود. گرچه پیچیدگی مکانیزم رسیدگی به استثنای پاسکال شیئی، آن را ناکارامد می‌کند و از این رو باید خردمندانه به کاربرده شود. در حالی که صدور استثناء تقریباً به هر دلیلی امکان‌پذیر بوده و حفاظت از هر بلوک کد با بسته‌بندی کردن آن در یک دستور **try...except** یا **try...finally** امکان‌پذیر است، در عمل این ابزار بهترین اندوخته برای موقعیت‌های به خصوص هستند.

مدیریت استثناء برای موارد زیر مناسب است:

- خطاهایی که شانس رخداد آنها پایین بوده یا دسترسی به آنها مشکل است، اما نتایج این خطاها احتمالاً مصیبت‌بار است (مانند قفل کردن برنامه).

---

۱ Exception

۲ Exception handler

۳ Inheritance

- برای شرایط خطایی که پیچیده هستند یا مشکل است تا آنها را در دستورات **if...then** بررسی کرد.
- و هر زمان که نیازمند عکس‌العمل نشان دادن به استثناهای صادره توسط سیستم عامل یا توسط روتین‌هایی که کد منبع آنها را کنترل نمی‌کنید، هستید.

استثناها به طور شایع برای خطاهای سخت‌افزار، I/O، حافظه و سیستم عامل به کار برده می‌شوند. دستورات شرطی اغلب بهترین راه برای بررسی خطاها هستند. برای مثال، فرض کنید که می‌خواهید مطمئن شوید که یک فایل قبل از تلاش برای باز کردنش موجود باشد. شما می‌توانید این کار را به روش زیر انجام دهید:

```
try
AssignFile(F, FileName);
Reset(F); // raises an EInOutError exception if file is not found
except
on Exception do ...
end;
```

اما ضمناً می‌توانید با استفاده از کد زیر از مخارج اضافی رسیدگی به استثناء هم پرهیز نمایید

```
if FileExists(FileName) then // returns False if file is not found; raises no exception
begin
AssignFile(F, FileName);
Reset(F);
end;
```

احکام[۱] شیوه دیگری از بررسی یک شرط بولی در هرجای کد منبع را عرضه می‌کنند. زمانی که یک دستور *Assert* نافرجام می‌ماند، برنامه یا متوقف می‌شود یا (اگر از یونیت *SysUtils* استفاده می‌کند) یک استثنای *EAssertionFailed* صادر می‌شود. بایستی احکام تنها برای بررسی شرایطی که انتظار ندارید که رخ دهند، به کار برده شوند.

اعلان نوع‌های استثناء

نوع‌های استثناء عیناً مانند کلاس‌های دیگر اعلان می‌شوند. در واقع، استفاده از یک وهله از هر کلاسی به عنوان یک استثناء امکان‌پذیر است، اما توصیه می‌شود که استثناها از کلاس *Exception* که در یونیت *SysUtils* تعریف شده است، مشتق شوند.

---

[۱] Assertions

شما می‌توانید با استفاده از توارث استثناها را در میان خانواده‌هایی گروه‌بندی کنید. برای مثال، اعلان‌های زیر در *SysUtils* خانواده‌ای از انواع استثناء را برای خطاهای ریاضیاتی تعریف می‌کنند.

```
type
EMathError = class(Exception);
EInvalidOp = class(EMathError);
EZeroDivide = class(EMathError);
EOverflow = class(EMathError);
EUnderflow = class(EMathError);
```

با این اعلان‌های داده شده، شما می‌توانید گرداننده استثناء *EMathError* واحدی را تعریف کنید که *EInvalidOp*، *EZeroDivide*، *EOverflow* و *EUnderflow* را هم اداره کند.

کلاس‌های استثناء برخی اوقات فیلدها، متدها یا خاصیت‌هایی را تعریف می‌کنند که اطلاعات اضافی را درباره خطا منتقل می‌کنند. برای مثال،

```
type EInOutError = class(Exception)
ErrorCode: Integer;
end;
```

تولید و اداره استثناها

برای ایجاد یک شیء استثناء، سازنده کلاس استثناء را در میان یک دستور **raise** فراخوانی کنید. برای مثال،

```
raise EMathError.Create;
```

به طور کلی، قالب یک دستور **raise** به صورت زیر است

```
raise object at address
```

جایی که *object* و *address* هردو اختیاری هستند. چنان چه از *object* صرف‌نظر شود، دستور استثناء جاری را بازتولید می‌کند؛ بخش «بازتولید استثناها» را در همین فصل ملاحظه نمایید. هرگاه یک *address* تعیین شده باشد، معمولاً اشاره‌گری به یک روال یا تابع خواهد بود؛ از این گزینه برای برانگیختن استثناء از یک نقطه قدیمی واقع در پشته نسبت به جایی که خطا واقعاً رخ داده است، استفاده کنید.

زمانی که یک استثناء تولید می‌شود— یعنی، در یک دستور **raise** ارجاع می‌شود — آن استثناء توسط یک منطق ویژه رسیدگی به استثناء کنترل می‌شود. دستور **raise** هرگز کنترل را به روش نرمال برگشت نمی‌دهد. در عوض کنترل را به درونی‌ترین گرداننده استثناء که قادر به اداره کردن استثناهای کلاس داده شده است، منتقل می‌کند. (درونی‌ترین گرداننده استثناء، گرداننده‌ای است که بلوک **try...except** آن زودتر از همه وارد شده اما هنوز خارج نشده است.) برای مثال، تابع زیر یک رشته را به عددی صحیح تبدیل می‌کند، با برانگیختن یک استثنای *ERangeError* در صورتی که مقدار نتیجه شده خارج از یک دامنه تعیین شده باشد.

```
function StrToIntRange(const S: string; Min, Max: Longint): Longint;
begin
Result := StrToInt(S);                    // StrToInt is declared in SysUtils
if (Result < Min) or (Result > Max) then
raise ERangeError.CreateFmt(
'%d is not within the valid range of %d..%d',
[Result, Min, Max]);
end;
```

توجه کنید که متد *CreateFmt* در دستور **raise** فراخوان شده است. *Exception* و فرزندانش سازنده‌های ویژه‌ای دارند که روش‌های جایگزینی برای ایجاد پیغام‌های خطا و IDهای متن، ارائه می‌دهند. یک استثنای تولید شده بعد از این که مدیریت شد، به طور خودکار تخریب می‌شود. هرگز سعی نکنید که یک استثنای تولید شده را به طور دستی تخریب کنید.

توجه صدور یک استثناء در بخش مقداردهی (initialization) یک یونیت ممکن است نتیجه مورد نظر را تولید نکند. پشتیبان استثنای نرمال از یونیت *SysUtils* می‌آید، که باید قبل از این که همچو پشتیبانی در دسترس باشد مقداردهی اولیه شود. اگر یک استثناء در طی ارزش‌گذاری (initialization) رخ دهد، به همه یونیت‌های مقداردهی شده — از جمله *SysUtils*— خاتمه داده شده و استثناء بازتولید می‌شود. سپس استثناء به طور معمول با وقفه در برنامه، دریافت شده و اداره می‌شود.

دستور Try...except

استثناها درمیان دستورات **try...except** اداره می‌شوند. برای مثال،

```
try
X := Y/Z;
except
```

```
on EZeroDivide do HandleZeroDivide;
end;
```

این دستور سعی می‌کند تا Y را بر X تقسیم کند، اما چنان چه یک استثناء *EZeroDivide* تولید شود، روتینی به نام *HandleZeroDivide* را فرامی‌خواند.

ترکیب نوشتاری/نحوی دستور **try...except** به این صورت است

```
try statements except exceptionBlock end
```

جایی که *statements* دنباله‌ای از دستورهاست (که توسط نقطه ویرگول‌ها از هم جدا می‌شوند) و *exceptionBlock* می‌تواند هریک از موارد زیر باشد

- دنباله دیگری از دستورات یا
- دنباله‌ای از گرداننده‌های استثناء، که به طور اختیاری با دستور زیر پی گرفته می‌شود

```
else statements
```

یک گرداننده اخطار فرم زیر را دارد

```
on identifier: type do statement
```

جایی که :*identifier* اختیاری بوده (اگر باشد، *identifier* می‌تواند هر شناسه معتبری باشد)، *type* یک نوع به کار رفته برای بیان استثناهاست و *statement* هر گونه دستوری می‌باشد.

دستور **try...except**، دستورات واقع در لیست *statements* اولیه را اجرا می‌کند. چنان چه هیچ استثنایی تولید نشود، از بلوک استثناء (*exceptionBlock*) صرف‌نظر شده و کنترل اجرا به بخش بعدی برنامه منتقل می‌شود.

چنان چه یک استثناء در طی اجرای لیست *statements* اولیه برانگیخته شود، در این صورت یا توسط یک دستور **raise** در لیست *statements* یا بوسیله یک روال یا تابع فراخوان شده از لیست *statements*، تلاشی برای اداره استثناء صورت می‌گیرد:

- اگر هر یک از گردانندهها در بلوک استثناء با استثناء مطابقت کند، کنترل به اولین گرداننده نظیر آن منتقل می‌شود. یک گرداننده استثناء یک استثناء را تنها در حالتی مطابقت می‌دهد کـه *type* در گرداننده، کلاس استثناء یا یک نیای آن کلاس باشد.
- اگر یک چنین گردانندهای پیدا نشود، کنترل به *statement* واقع در شرط **else** منتقل می‌شـود، چنان چه شرط **else**ای در آنجا موجود باشد.
- اگر بلوک استثناء صرفاً دنبالهای از دستورات بدون هر گونه گرداننده اخطاری باشد، کنترل به اولین دستور واقع در لیست منتقل می‌شود.
- اگر هیچ کدام از شرایط بالا ارضاء نشوند، جستجو در بلـوک اسـتثنای دسـتور **try...except** بعدی که دیرتر از همه وارد شده و هنوز خارج نشده، ادامه می‌یابد. اگر هیچ گرداننده مناسبی، شرط **else**، یا لیست دستوری در آنجا یافت نشود، به دستور **try...except** بعدی که دیرتر از همه وارد شده، انتقال می‌یابد و الی آخر. اگر بیرونی‌ترین دستور **try...except** به دست آمـده باشد و استثناء هنوز مدیریت نشده باشد، اجرای برنامه خاتمه می‌یابد.

هر گاه یک استثنا مدیریت شود، پشته بـه عقـب و بـه روال یـا تـابع محتـوی دسـتور **try...except** برمی‌گردد جایی که رسیدگی به استثناء ظاهر می‌شود و کنترل به گردانندهٔ اسـتثنای اجـرا شـده، شـرط **else** یا لیست دستور، منتقل می‌شود. این فرایند تمامی فراخوان‌های توابع و روال‌ها را که بعـد از وارد کردن دستور **try...except** جایی که استثناء اداره می‌شود، رخ دادهانـد، دور مـی‌انـدازد. سـپس شـیء استثناء به طور خودکار از طریق یک فراخوان به تخریب کننده *Destroy*، از بین رفته و کنترل به دسـتور بعد از دستور **try...except** منتقل می‌شود. (چنان چه یک فراخوان به روال اسـتاندارد *Exit*، *Break* یـا *Continue* باعث شود که کنترل گردانندهٔ استثناء را ترک کند، شیء استثناء بـاز هـم بـه طـور خودکـار تخریب می‌شود.)

در مثال زیر، اولین گرداننده استثناء، اسـتثناهای تقسـیم بـر صـفر را اداره مـی‌کنـد، دومـین گردانـنده استثناهای سرریز را اداره می‌کند، و گردانندهٔ نهایی تمامی‌استثناهای ریاضیاتی دیگـر را اداره مـی‌کنـد. *EMathError* آخر از همه در بلوک استثناء ظاهر می‌شود زیرا نیای دو کلاس استثنای دیگر است؛ چنان چه در ابتدا ظاهر شود، دو گردانندهٔ دیگر هرگز احضار نخواهند شد.

```
try
...
except
```

```
on EZeroDivide do HandleZeroDivide;
on EOverflow do HandleOverflow;
on EMathError do HandleMathError;
end;
```

یک گرداننده استثناء می‌تواند قبل از اسم کلاس استثناء، یک شناسه تعیین کند. این کار شناسه‌ای را اعلان می‌کند که بیانگر شیء استثنایی است که در طی اجرای دستوری که بعد از **on...do** می‌آید، رخ می‌دهد. سطح دسترسی شناسه محدود به آن دستور است. برای مثال،

```
try
...
except
on E: Exception do ErrorDialog(E.Message, E.HelpContext);
end;
```

اگر بلوک استثناء یک شرط **else** را تصریح کند، شرط **else** هر استثنایی را که توسط گردانندەهای استثنای بلوک اداره نشده‌اند، اداره می‌کند. برای مثال،

```
try
...
except
on EZeroDivide do HandleZeroDivide;
on EOverflow do HandleOverflow;
on EMathError do HandleMathError;
else
HandleAllOthers;
end;
```

در اینجا، شرط **else** هر استثنایی را که یک *EMathError* نباشد اداره می‌کند.

یک بلوک استثناء که دارای هیچ گرداننده استثنایی نباشد، اما در عوض تنها از لیستی از دستورها تشکیل شده باشد، همه استثناها را اداره می‌کند. برای مثال،

```
try
...
except
HandleException;
end;
```

در این جا، روتین *HandleException* هر استثنایی را که به عنوان نتیجه اجرای دستورات واقع در میان **try** و **except** اتفاق می‌افتد، اداره می‌کند.

بازتولید استثناها

هرگاه واژه کلیدی **raise** در یک بلوک استثناء بدون یک ارجاع شیء بعد از آن، رخ دهد، هر استثنایی را که توسط بلوک اداره شده باشد بازتولید می‌کند. این امر به یک گرداننده استثناء اجازه می‌دهد که به یک خطا با روشی محدود شده عکس‌العمل نشان دهد و سپس استثناء را بازتولید کند. زمانی که یک روال یا تابع بعد از رخداد یک استثناء لازم است که پاک‌سازی شود اما نمی‌تواند به طور کامل استثناء را اداره کند، بازتولید استثناء سودمند و پرفایده خواهد بود. برای مثال، تابع *GetFileList* یک شیء *TStringList* را تخصیص حافظه داده و آن را با اسامی فایل مطابق با یک مسیر جستجوی تعیین شده، پر می‌کند:

```
function GetFileList(const Path: string): TStringList;
var
I: Integer;
SearchRec: TSearchRec;
begin
Result := TStringList.Create;
try
I := FindFirst(Path, 0, SearchRec);
while I = 0 do
begin
Result.Add(SearchRec.Name);
I := FindNext(SearchRec);
end;
except
Result.Free;
raise;
end;
end;
```

GetFileList یک شیء *TStringList* ایجاد می‌کند، سپس از توابع *FindFirst* و *FindNext* (که در یونیت *SysUtils* تعریف شده‌اند) استفاده می‌کند تا شیء را مقداردهی اولیه کند. چنان چه مقداردهی ناتمام بماند— برای مثال اگر مسیر جستجو نامعتبر باشد، یا اگر در آن جا حافظه کافی برای پر کردن لیست رشته وجود نداشته باشد— *GetFileList* نیازمند آزادسازی لیست رشته جدید است، از آن جایی که فراخواننده هنوز از وجود او خبر ندارد. به این دلیل، مقداردهی لیست رشته در یک دستور **try...except** انجام می‌شود. اگر یک استثناء رخ دهد، بلوک استثناء دستور لیست رشته را آزاد کرده سپس استثناء بازتولید می‌شود.

استثناهای تودرتو

کد اجرا شده در یک گرداننده استثناء، خود می‌تواند استثناها را برانگیخته و اداره کند. در ضمن تا زمانی که این استثناها در میان گرداننده استثناء اداره شوند، آنها بر استثنای اصلی تأثیری نمی‌گذارند.

گرچه همین که یک استثنای تولید شده در یک گرداننده استثناء به ماورای آن گردانـنده منتشـر شـود، استثنای اصلی مفقود می‌شود. در کد زیر این امر توسط تابع *Tan* تشریح شده است.

```
type
ETrigError = class(EMathError);
function Tan(X: Extended): Extended;
begin
try
Result := Sin(X) / Cos(X);
except
on EMathError do
raise ETrigError.Create('Invalid argument to Tan');
end;
end;
```

اگر یک استثنای *EMathError* در طی اجرای *Tan* اتفاق بیفتد، گرداننده استثناء یـک *ETrigError* تولیـد می‌کند. از آن جایی که *Tan* گردانندهای برای *ETrigError* فراهم نمی‌کند، استثناء بـه مـاورای گردانـنده استثنای اصلی منتقل می‌شود و باعث می‌شود که استثنای *EMathError* تخریب گردد. ایـن اتفـاق بـرای فراخواننده، طوری رخ می‌دهد که انگار تابع *Tan* یک استثنای *ETrigError* تولید کرده است.

دستور Try...finally

برخی اوقات شما می‌خواهید مطمئن شوید که بخش‌های خاصی از یک عملیات کامل شده است، خـواه عملیات با یک استثناء متوقف شده باشد یا نه. برای مثال، زمانی که یک روتین کنترل یـک منبـع را بـه دست می‌گیرد، اغلب مهم است که منبع آزاد شود، علیرغم این که آیا روتین به طور نرمال خاتمه یافتـه باشد یا نه. در این گونه وضعیت‌ها، شما می‌توانید از دستور **try...finally** استفاده کنید.

مثال زیر نشان می‌دهد که چگونه کدی که یک فایل را باز کرده و پردازش می‌کند، می‌تواند مطمئن شود که فایل نهایتاً بسته می‌شود، حتی اگر یک خطا در طی اجرا رخ دهد.

```
Reset(F);
try
  ...                  // process file F
finally
  CloseFile(F);
end;
```

ترکیب نوشتاری/نحوی دستور **try...finally** به صورت زیر است

```
try statementList1 finally statementList2 end
```

جایی که هر *statementList* دنباله‌ای از دستورات است که توسط نقطه ویرگول از هم جدا می‌شوند. دستور **try...finally** دستورات *statementList*1 (شرط **try**) را اجرا می‌کند. اگر *statementList*1 بدون صدور استثنایی پایان یابد، *statementList*2 (شرط **finally**) اجرا می‌شود. اگر یک استثناء در طی اجرای *statementList*1 تولید شود، کنترل برنامه به *statementList*2 منتقل می‌شود؛ همین که *statementList*2 به اجرا شدن خاتمه دهد، استثناء مجدداً تولید می‌شود. چنان چه یک فراخوان به روال *Exit*، *Break* یا *Continue* باعث شود که کنترل برنامه *statementList*1 را ترک کند، *statementList*2 به طور خودکار اجرا می‌شود. از این رو شرط **finally** همواره اجرا می‌شود، علیرغم این که شرط **try** چگونه خاتمه یافته باشد.

چنان چه یک استثناء تولید شود اما در شرط **finally** اداره نشود، آن استثناء به خارج از دستور **try...finally** منتشر می‌شود و هر استثنایی که قبلاً در شرط **try** صادر شده بود مفقود می‌شود. از این رو برای مختل نکردن انتشار استثناهای دیگر، شرط **finally** بایستی همه استثناهایی را که به طور محلی صادر می‌شوند، اداره کند.

کلاس‌ها و روتین‌های استثنای استاندارد

یونیت *SysUtils* روتین‌های استاندارد متعددی شامل *ExceptObject*،*ExceptAddr* و *ShowException* برای اداره کردن استثناها اعلان می‌کند. در ضمن *SysUtils* و یونیت‌های دیگر شامل چند دوجین کلاس استثناء هستند، که همگی آنها (قطع نظر از *OutlineError*) از *Exception* مشتق می‌شوند.

کلاس *Exception* خاصیت‌هایی به نام *Message* و *HelpContext* دارد که می‌توانند برای ارسال یک توضیح خطا و یک ID متن برای مستندات درون خطی حساس نسبت به متن، به کار برده شوند. در ضمن این کلاس متدهای سازنده گوناگونی تعریف می‌کند که به شما اجازه می‌دهند تا توضیح و ID متن را به روش‌های مختلفی تصریح نمایید.

فصل ۸

روتین‌های استاندارد و I/O

این فصل درباره متن و I/O فایل بحث کرده و روتین‌های کتابخانه‌ای استاندارد را جمع‌بندی می‌کند. خیلی از روال‌ها و توابع فهرست شده در اینجا در یونیت *System* تعریف شده‌اند؛ این یونیت به طور ضمنی با هر برنامه‌ای کامپایل می‌شود. بقیه در میان کامپایلر ساخته می‌شوند اما طوری با آنها رفتار می‌شود که انگار در یونیت *System* بوده‌اند.

برخی روتین‌های استاندارد در یونیت‌هایی مانند *SysUtils* قرار دارند، که این یونیت‌ها بایستی در یک شرط **uses** لیست شوند تا آنها را در دسترس برنامه‌ها قرار دهد. گر چه، شما نمی‌توانید *System* را در یک شرط **uses** لیست کنید، یا نباید یونیت *System* را ویرایش کنید یا سعی نمایید که آن را به طور ضمنی از نو ایجاد کنید.

ورودی و خروجی فایل

جدول زیر روتین‌های ورودی و خروجی را فهرست‌بندی می‌کند.

Table 8.1 روال‌ها و توابع **Input** و **Output**

تابع	توضیح

Append	یک فایل متنی موجود را برای الحاق کردن باز می‌کند.
AssignFile	نام یک فایل بیرونی را به یک متغیر فایل تخصیص می‌دهد.
BlockRead	یک یا چند رکورد را از یک فایل بدون نوع می‌خواند.
BlockWrite	یک یا چند رکورد را درون یک فایل بدون نوع می‌نویسد.
ChDir	دایرکتوری جاری را تغییر می‌دهد.
CloseFile	یک فایل باز را می‌بندد.
Eof	وضعیت انتهای پرونده یک فایل را برمی‌گرداند.
Eoln	وضعیت انتهای سطر یک فایل متنی را برمی‌گرداند.
Erase	یک فایل بیرونی را پاک می‌کند.
FilePos	موقعیت فایل جاری یک فایل بدون نوع یا نوع‌دار را برمی‌گرداند.
FileSize	اندازه جاری یک فایل را برمی‌گرداند؛ برای فایل‌های متنی به کار نمی‌رود.
Flush	بافر یک فایل متنی خروجی را خالی می‌کند.
GetDir	دایرکتوری جاری یک درایو مشخص شده را برمی‌گرداند.
IOResult	یک مقدار صحیح برمی‌گرداند که وضعیت آخرین تابع I/O انجام شده است.
MkDir	یک زیر دایرکتوری ایجاد می‌کند.
Read	یک یا چند مقدار را از یک فایل به درون یک یا چند متغیر می‌خواند.
Readln	آن چه را که *Read* انجام می‌دهد، انجام داده و به ابتدای خط بعدی درفایل متنی می‌رود.
Rename	یک فایل بیرونی را تغییر نام می‌دهد.
Reset	یک فایل موجود را باز می‌کند.
Rewrite	یک فایل جدید را ایجاد کرده و باز می‌کند.
RmDir	یک زیردایرکتوری خالی را حذف می‌کند.
Seek	موقعیت جاری یک فایل نوع‌دار یا بدون نوع را به جزء مشخصی جابه‌جا می‌کند. برای فایل‌های متنی به کار نمی‌رود.
SeekEof	وضعیت انتهای فایل یک فایل متنی را برمی‌گرداند.
SeekEoln	وضعیت انتهای سطر یک فایل متنی را برمی‌گرداند.
SetTextBuf	یک بافر I/O را به یک فایل متنی تخصیص می‌دهد.
Truncate	یک فایل نوع‌دار یا بدون نوع را در موقعیت جاری فایل کوتاه می‌کند.
Write	یک یا چند مقدار را به یک فایل می‌نویسد.
Writeln	همانند *Write* عمل می‌کند و سپس یک نشانگر انتهای سطر به فایل متنی می‌نویسد.

یک متغیر فایل هر متغیری است که نوعش یک نوع فایل باشد. در این جا سه کلاس از فایل‌ها وجود دارد: نوع‌دار (*typed*)، متنی (*text*) و بدون نوع (*untyped*). ترکیب نوشتاری/نحوی برای اعلان انواع فایل در بخش «انواع فایل» در فصل۵ داده شده است.

قبل از این که یک متغیر فایل بتواند به کار گرفته شود، باید از طریق یک فراخوان به روال *AssignFile* با یک فایل بیرونی مرتبط شود. فایل بیرونی نوعاً یک فایل دیسک مشخص است، اما می‌تواند یک ابزار نیز باشد، مانند صفحه کلید یا نمایشگر. فایل بیرونی اطلاعات نوشته شده به فایل را ذخیره می‌کند یا اطلاعات خوانده شده از فایل را عرضه می‌کند.

همین که ارتباط با فایل بیرونی برقرار گردد، متغیر فایل بایست باز شود تا آن را برای ورودی و خروجی آماده کند. یک فایل موجود می‌تواند از طریق روال *Reset* باز شود و یک فایل جدید می‌تواند از طریق روال *Rewrite* ایجاد شده و باز گردد.

فایل‌های متنی باز شده با *Reset* فقط–خواندنی هستند و فایل‌های متنی باز شده با *Rewrite* و *Append* فقط–نوشتنی هستند. فایل‌های نوعدار و فایل‌های بدون نوع همواره مجاز هستند که هم خوانده شده و هم نوشته شوند، علیرغم این که آنها با *Reset* باز شده باشند یا با *Rewrite*.

هر فایل دنباله‌ای خطی از اجزاست، که هر کدام از آنها نوع جزء (یا نوع رکورد) فایل را دارند. اجزا با شروع از صفر شماره‌گذاری می‌شوند.

فایل‌ها معمولاً به طور ترتیبی قابل دسترسی می‌باشند. یعنی، زمانی که یک جزء با استفاده از روال استاندارد *Read* خوانده شده و یا با استفاده از روال استاندارد *Write* نوشته می‌شود، موقعیت جاری فایل به جزء فایل مرتب شده بعدی از حیث عدد حرکت می‌کند. در ضمن فایل‌های نوعدار و فایل‌های بدون نوع می‌توانند از طریق روال استاندارد *Seek* به طور تصادفی قابل دسترسی باشند، که این روال موقعیت جاری فایل را به جزء مشخص شده‌ای جابه‌جا می‌کند. توابع استاندارد *FilePos* و *FileSize* می‌توانند برای تشخیص موقعیت جاری فایل و اندازه فعلی فایل به کار برده شوند.

هرگاه یک برنامه پردازش یک فایل را کامل می‌کند، بایستی فایل با استفاده از روال استاندارد *CloseFile* بسته شود. بعد از این که یک فایل بسته شد، فایل بیرونی مرتبط با او به روز می‌شود. پس از آن متغیر فایل می‌تواند با فایل بیرونی دیگری مرتبط گردد.

به طور پیش فرض، هر فراخوانی بـه روال‌ها و توابع I/O به طور خودکـار بـه منظـور یـافتن خطاهـا بررسی می‌شود و چنان چه یک خطا اتفاق بیفتد یک استثناء تولید می‌شود (یا اگـر اداره اسـتثناء فعـال نشده باشد برنامه خاتمه می‌یابد). این بررسی خودکار مـی‌توانـد بـا اسـتفاده از راهنمـاهـای کامپـایلر **{$I+}** و **{$I–}** فعال و غیرفعال شود. هرگاه بررسی I/O غیر فعال باشد— یعنی فراخوان روال یـا تابع در حالت **{$I–}** کامپایل شود— یک خطای I/O سبب تولید یک استثناء نخواهد شد؛ در عـوض برای بررسی نتیجه یک عملیات I/O ، شما بایستی تابع استاندارد *IOResult* را فراخوانی کنید.

شما باید تابع *IOResult* را برای پاک کردن یک خطا فراخوانی کنید، حتی اگر مایل به خطا نباشید. اگر یک خطا را پاک نکنید و **{$I+}** حالت جاری کامپایل باشد، فراخوان بعدی تابع I/O شما با خطـای ممتد *IOResult* نافرجام خواهد ماند.

فایل‌های متنی

این بخش استفاده متغیرهای فایل از نوع استاندارد *Text* را جمع‌بندی می‌کند. هرگاه یک فایل متنی بـاز می‌شود، فایل بیرونی به روش خاصی تفسـیر مـی‌شـود: ایـن فایـل بـه صـورت نمـایش دنبالـه‌ای از کاراکترهای فرمت شده در سطرها مورد توجه قرار می‌گیرد، جایی که هـر سـطر توسـط یـک نشـانگر انتهای سطر (یک کاراکتر بازگشت نورد (carriage-return)، که احتمالاً با یـک کـاراکتر تعـویض سـطر (line-feed)پی گرفـته می‌شود). خاتمه می‌یابد. نوع *Text* مجزا از نوع file of Char می‌باشد.

برای فایل‌های متنی، شکل‌های خاصی از *Read* و *Write* موجود است که بـه شـما اجـازه خوانـدن و نوشتن مقادیری را می‌دهند که از نوع *Char* نیستند. یک چنین مقادیری به طور خودکار از/بـه نمـایش کاراکتریشان ترجمه می‌شوند. بـرای مثـال، Read(F, I)، جـایی کـه I متغیـری از نـوع *Integer* اسـت، دنباله‌ای از ارقام را می‌خواند، دنباله را به صورت یک عدد صـحیح دهـدهی تعبیـر کـرده و آن را در I ذخیره می‌کند.

دو متغیر فایل متنی استاندارد موجود است، *Input* و *Output*. متغیر فایـل اسـتاندارد *Input* یـک فایـل فقط خواندنی مرتبط با ورودی استاندارد سیستم عامل (برای نمونه، صـفحه کلیـد) اسـت. متغیـر فایـل استاندارد *Output* یک فایل فقط نوشتنی مرتبط بـا خروجـی اسـتاندارد سیسـتم عامـل (بـرای نمونـه

نمایشگر) است. قبل از این که یک برنامه شروع به اجرا کند، *Input* و *Output* به طور خودکار باز می‌شوند، به طوری که انگار دستورات زیر اجرا شده باشند:

```
AssignFile(Input, '');
Reset(Input);
AssignFile(Output, '');
Rewrite(Output);
```

توجه I/O متن‌گرا تنها در برنامه‌های کنسول در دسترس هستند— یعنی، برنامه‌های کامپایل شده با گزینه "Generate console application" که در صفحه Linker جعبه محاوره‌ای Project Options انتخاب شده است یا با گزینه کامپایلر خط فرمان **-cc**. در یک برنامه GUI (غیر کنسول)، هر گونه تلاشی برای خواندن یا نوشتن با استفاده از *Input* یا *Output* یک خطای I/O تولید خواهد کرد.

برخی از روتین‌های استاندارد I/O که روی فایل‌های متنی کار می‌کنند نیازی به یک متغیر فایل که به طور صریح به صورت یک پارامتر داده شده باشد، ندارند. اگر از پارامتر فایل صرف‌نظر شود، به طور پیش *Input* یا *Output* در نظر گرفته می‌شود بسته به این که تابع یا روال ورودی‌گرا یا خروجی‌گرا باشند. برای مثال، Read(X) متناظر با Read(Input, X) و Write(X) متناظر با Write(Output, X) است.

چنان چه فایلی را هنگام فراخوانی یکی از روتین‌های ورودی یا خروجی که روی فایل‌های متنی کار می‌کند، تعیین کنید، فایل باید با استفاده از *AssignFile* با یک فایل بیرونی مرتبط شود و با استفاده از *Reset*، *Rewrite* یا *Append* باز شود. چنان چه فایلی را که با *Reset* باز شده است به یک روال یا تابع خروجی‌گرا ارسال کنید، یک استثناء تولید می‌شود. در ضمن اگر فایلی را که با *Rewrite* یا *Append* باز شده است به یک روال یا تابع ورودی‌گرا ارسال کنید، یک استثناء تولید می‌شود.

فایل‌های بدون نوع

فایل‌های بدون نوع، کانال‌های I/O تراز پایینی هستند که اصولاً علیرغم نوع و ساختاربندی برای دسترسی مستقیم به فایل‌های دیسک به کار برده می‌شوند. یک فایل بدون نوع با واژه **file** و نه هیچ چیز بیشتری اعلان می‌شود. برای مثال،

```
var DataFile: file;
```

برای فایل‌های بدون نوع، روال‌های *Reset* و *Rewrite* اجازه استفاده از یک پارامتر اضافی برای تعیین اندازه رکورد به کار رفته در انتقال داده‌ها را می‌دهند. به دلایل تاریخی، اندازه رکورد پیش فرض ۱۲۸ بایت است. یک رکورد از اندازه ۱ تنها مقداری است که به درستی اندازه دقیق هر فایل را بازتاب می‌دهد. (هرگاه اندازه رکورد یک باشد، هیچ رکورد جزئی امکان‌پذیر نخواهد بود.)

به استثنای *Read* و *Write*، همه روال‌ها و توابع استاندارد فایل نوعدار روی فایل‌های بدون نوع نیز مجاز هستند. به جای *Read* و *Write*، دو روال به نام‌های *BlockRead* و *BlockWrite* برای نقل و انتقالات داده با سرعت بالا به کار برده می‌شوند.

گرداننده‌های ابزار فایل متنی

شما می‌توانید گرداننده‌های ابزار فایل متنی[۱] مال خود را برای برنامه خود تعریف کنید. یک گرداننده ابزار فایل متنی مجموعه‌ای متشکل از چهار تابع است که به طور کامل یک واسط (interface) میان سیستم فایلی پاسکال شیئی و برخی ابزارها را پیاده‌سازی می‌کنند.

چهار تابعی که هر گرداننده ابزار را تعریف می‌کنند، *Open*، *InOut*، *Flush* و *Close* هستند. هدر هر تابع به صورت زیر است

```
function DeviceFunc(var F: TTextRec): Integer;
```

جایی که *DeviceFunc* اسم تابع است (یعنی *Open*، *InOut*، *Flush* یا *Close*). برای آگاهی از اطلاعات بیشتر درباره نوع *TTextRec* راهنمای درون خطی دلفی را ملاحظه نمایید. مقدار برگشتی یک تابع رابط–ابزار، مقدار برگردانده شده توسط *IOResult* می‌شود. چنان چه مقدار برگشتی صفر باشد، عملیات موفقیت‌آمیز بوده است.

برای مرتبط کردن توابع رابط–ابزار با یک فایل مشخص، شما باید یک روال سفارشی شده *Assign* بنویسید. روال *Assign* باید آدرس‌های چهار تابع رابط–ابزار را به چهار اشاره‌گر تابع در متغیر فایل متنی، تخصیص دهد. علاوه بر این، این روال باید ثابت جادویی *fmClosed* را در فیلد *Mode* ذخیره

[۱] Text-file Device Drivers

کند، اندازه بافر فایل متنی را در *BufSize* ذخیره کند، یک اشاره‌گر به بافر فایل متنی را در *BufPtr* ذخیره کند و رشته *Name* را پاک نماید.

برای مثال، با فرض این که چهار تابع رابط–ابزار *DevOpen*، *DevInOut*، *DevFlush* و *DevClose* خوانده شوند، روال *Assign* ممکن است به این صورت به نظر رسد:

```
procedure AssignDev(var F: Text);
begin
with TTextRec(F) do
begin
Mode := fmClosed;
BufSize := SizeOf(Buffer);
BufPtr := @Buffer;
OpenFunc := @DevOpen;
InOutFunc := @DevInOut;
FlushFunc := @DevFlush;
CloseFunc := @DevClose;
Name[0] := #0;
end;
end;
```

توابع رابط–ابزار می‌توانند از فیلد *UserData* واقع در رکورد فایل برای ذخیره اطلاعات شخصی استفاده کنند. این فیلد به هیچ وجه توسط سیستم فایل محصول ویرایش نمی‌شود.

توابع ابزار

توابعی که یک گرداننده ابزار فایل متنی را ایجاد می‌کنند در زیر تشریح شده اند.

تابع Open

تابع *Open* توسط روال‌های استاندارد *Reset*، *Rewrite* و *Append* برای باز کردن یک فایل متنی مرتبط با یک ابزار، فراخوانده می‌شود. در مدخل، فیلد *Mode*، از *fmInput*، *fmOutput* یا *fmInOut* تشکیل می‌شود تا نشان دهد که آیا تابع *Open* از *Reset*، *Rewrite* یا *Append* فراخوان شده بود.

تابع *Open* فایل را بر طبق مقدار *Mode*، برای ورودی و خروجی آماده می‌کند. اگر *Mode*، *fmInOut* را مشخص کرده باشد (نشانگر این که *Open* از *Append* فراخوان شده بود)، بایستی *Mode* قبل از این که *Open* برگردد، به *fmOutput* تغییر داده شود.

Open همواره قبل از هر یک از توابع رابط–ابزار دیگر فراخوانده می‌شود. به همین دلیل، *AssignDev* تنها فیلد *OpenFunc* را مقداردهی می‌کند، با ترک مقداردهی بردارهای باقیمانده تا حدود *Open*. بر

اساس *Mode*، آنگاه تابع *Open* می‌تواند اشاره‌گرهایی را برای توابع ورودی‌گرا یا خروجی‌گرا نصب کند. این کار توابع *InOut* و *Flush* و روال *CloseFile* را از تشخیص مد جاری بی‌نیاز می‌کند.

تابع InOut

تابع *InOut* توسط روتین‌های استاندارد *Read*، *Readln*، *Write*، *Writeln*، *Eof*، *Eoln*، *SeekEof*، *SeekEoln* و *CloseFile* چنان چه ورودی یا خروجی از ابزار لازم باشد، فراخوان می‌شود.

هرگاه *Mode* برابر *fmInput* باشد، تابع *InOut* کاراکترها را به میزان *BufSize* در BufPtr^ قرائت می‌کند و تعداد کاراکترهای خوانده شده در *BufEnd* را برمی‌گرداند. علاوه بر این، صفر را در *BufPos* ذخیره می‌کند. چنان چه تابع *InOut* صفر را در *BufEnd* به صورت یک نتیجه برای یک درخواست ورودی برگرداند، *Eof* برای فایل *True* می‌گردد. هرگاه *Mode* برابر *fmOutput* باشد، تابع *InOut* کاراکترهای *BufPos* را از BufPtr^ می‌نویسد، و صفر را در *BufPos* برمی‌گرداند.

تابع Flush

تابع *Flush* در انتهای هر *Read*، *Readln*، *Write* و *Writeln* فراخوان می‌شود. این تابع به طور انتخابی بافر فایل متنی را خالی می‌کند. اگر *Mode* برابر *fmInput* باشد، تابع *Flush* می‌تواند صفر را در *BufPos* و *BufEnd* ذخیره کند تا کاراکترهای باقیمانده (قرائت نشده) در بافر را خالی کند. این ویژگی به ندرت به کار برده می‌شود. اگر *Mode* برابر *fmOutput* باشد، تابع *Flush* می‌تواند محتوای بافر را دقیقاً مانند تابع *InOut* بنویسد، تا مطمئن شود که متن نوشته در ابزار، فوراً در روی ابزار ظاهر می‌شود. اگر *Flush* هیچ کاری انجام ندهد، متن تا زمانی که بافر پر شود یا فایل بسته شود، روی ابزار ظاهر نمی‌شود.

تابع Close

تابع *Close* توسط روال استاندارد *CloseFile* فراخوانده می‌شود تا یک فایل متنی مرتبط با یک ابزار را ببندد. (روال‌های *Reset*، *Rewrite* و *Append* نیز چنان چه فایلی را که در حال بازکردنش هستند قبلاً باز شده باشد، *Close* را فرامی‌خوانند.) اگر *Mode* برابر *fmOutput* باشد، در این صورت قبل از فراخوانی *Close*، سیستم تابع *InOut* را فراخوان می‌کند تا مطمئن شود که همه کاراکترها به ابزار نوشته شده‌اند.

اداره کردن رشته‌های منتهی به تهی

ترکیب نوشتاری/نحوی بسط یافته پاسکال شیئی اجازه می‌دهد تا روال‌های استاندارد *Read*، *Readln*، *Str* و *Val* به آرایه‌های کاراکتری پایه صفر اعمال شوند و ضمناً اجازه می‌دهد تا روال‌های استاندارد *Write*، *Writeln*، *Val*، *AssignFile* و *Rename* هم به آرایه‌های کاراکتری پایه صفر و هم به اشاره‌گرهای کاراکتری اعمال شوند. علاوه بر این، توابع زیر برای اداره کردن رشته‌های منتهی به تهی[1] عرضه شده‌اند. برای آگاهی از اطلاعات بیشتر درباره رشته‌های منتهی به تهی بخش «کار با رشته‌های منتهی به تهی» را در فصل۵ ملاحظه نمایید.

توابع رشته منتهی به تهی **Table 8.2**

تابع	توضیح
StrAlloc	یک بافر کاراکتر از اندازه داده شده‌ای را روی Heap منظور می‌کند.
StrBufSize	اندازه یک بافر کاراکتر تخصیص یافته با استفاده از *StrAlloc* یا *StrNew* را برمی‌گرداند.
StrCat	دو رشته را به هم می‌چسباند.
StrComp	دو رشته را مقایسه می‌کند.
StrCopy	یک رشته را کپی می‌کند.
StrDispose	یک بافر کاراکتر تخصیص یافته با استفاده از *StrAlloc* یا *StrNew* را آزاد می‌کند.
StrECopy	یک رشته را کپی کرده و اشاره‌گر را به انتهای رشته برمی‌گرداند.
StrEnd	یک اشاره‌گر به انتهای رشته را برمی‌گرداند.
StrFmt	یک یا چند مقدار را به درون یک رشته فرمت می‌کند.
StrIComp	دو رشته را بدون حساسیت نسبت به حالت حروف مقایسه می‌کند.
StrLCat	دو رشته را با یک طول حداکثری داده شده از رشته نتیجه، به هم متصل می‌کند.
StrLComp	دو رشته را برای یک طول حداکثری داده شده مقایسه می‌کند.
StrLCopy	یک رشته را تا یک طول حداکثری داده شده کپی می‌کند.
StrLen	طول یک رشته را برمی‌گرداند.
StrLFmt	یک یا چند مقدار را به یک رشته با یک طول حداکثری داده شده، فرمت می‌کند.

[1] Null-terminated strings

StrLIComp	دو رشته را برای یک طول حداکثری داده شده، بدون حساسیت نسبت به حالت حروف، مقایسه می‌کند.
StrLower	یک رشته را به رشته‌ای با حروف کوچک تبدیل می‌کند.
StrMove	بلوکی از کارکترها را از یک رشته به رشته‌ای دیگر جابه‌جا می‌کند.
StrNew	یک رشته را روی heap تخصیص حافظه می‌دهد.
StrPCopy	یک رشته پاسکال را به یک رشته منتهی به تهی کپی می‌کند.
StrPLCopy	یک رشته پاسکال را به یک رشته منتهی به تهی با یک طول حداکثری داده شده کپی می‌کند.
StrPos	اشاره‌گری را به اولین رخداد یک زیر رشته داده شده در میان یک رشته برمی‌گرداند.
StrRScan	اشاره‌گری را به آخرین رخداد یک زیر رشته داده شده در میان یک رشته برمی‌گرداند.
StrScan	اشاره‌گری را به اولین رخداد یک کاراکتر داده شده در میان یک رشته برمی‌گرداند.
StrUpper	یک رشته را به رشته‌ای با حروف بزرگ تبدیل می‌کند.

توابع استاندارد اداره رشته‌ها شرکای مولتی بایت فعالی دارند که مرتب‌سازی ویژه–منطقه را هم برای کاراکترها پیاده‌سازی می‌کنند. اسامی توابع مولتی بایت با *Ansi-* شروع می‌شوند. برای مثال، نسخه مولتی بایت *StrPos*، تابع *AnsiStrPos* است. پشتیبانی کاراکتر مولتی بایت وابسته به سیستم عامل و مبنی بر منطقه جاری است.

رشته‌های کاراکتر پهن

یونیت *System* سه تابع *WideCharToString*، *WideCharLenToString* و *StringToWideChar* را مهیا کرده است که می‌توانند برای تبدیل رشته‌های کاراکتر پهن منتهی به تهی به رشته‌های بلند تک بایت یا بایت مضاعف به کار برده شوند. برای آگاهی از اطلاعات بیشتر درباره رشته‌های کاراکتر پهن، بخش «بحثی پیرامون مجموعه کاراکترهای بسط یافته» را در فصل۵ ملاحظه نمایید.

روتین‌های استاندارد دیگر

جدول زیر روال‌ها و توابعی که بارها به کار رفته و در کتابخانه‌های محصول بورلند یافت می‌شوند، فهرست می‌کند. این یک فهرست جامع و فراگیر از روتین‌های استاندارد نیست. برای آگاهی از اطلاعات بیشتر درباره این روتین‌ها و روتین‌های دیگر راهنمای درون خطی دلفی را ملاحظه نمایید.

روتین‌های استاندارد دیگر **Table 8.3**

روال یا تابع	توضیح
Abort	پردازش را بدون گزارش کردن یک خطا به پایان می‌رساند.
Addr	اشاره‌گری را به شیء تعیین شده برمی‌گرداند.
AllocMem	بلوکی از حافظه را تخصیص داده و هر بایت را با صفر مقداردهی می‌کند.
ArcTan	آرک تانژانت عدد داده شده را برمی‌گرداند.
Assert	بررسی می‌کند که آیا یک عبارت بولی *True* است یا نه.
Assigned	برای یک اشاره‌گر **nil** (unassigned) یا متغیر رویه‌ای بررسی انجام می‌دهد.
Beep	بوق استانداردی را با استفاده از بلندگوی کامپیوتر تولید می‌کند.
Break	باعث می‌شود که کنترل برنامه از یک دستور **for**، **while** یا **repeat** خارج شود.
ByteToCharIndex	موقعیت کاراکتر محتوی یک بایت مشخص را در یک رشته برمی‌گرداند.
Chr	کاراکتری را برای یک مقدار مشخص برمی‌گرداند.
Close	به ارتباط مابین یک متغیر فایل و یک فایل بیرونی خاتمه می‌دهد.
CompareMem	یک مقایسه باینری از دو تصویر حافظه انجام می‌دهد.
CompareStr	رشته‌ها را با حساسیت نسبت به حالت حروف مقایسه می‌کند.
CompareText	رشته‌ها را با مقدار ترتیبی مقایسه می‌کند و به حالت حروف حساس نیست.
Continue	کنترل را به تکرار بعدی دستورات **for**، **while** یا **repeat** می‌برد.
Copy	زیر رشته‌ای از یک رشته یا قطعه‌ای از یک آرایه پویا را برمی‌گرداند.
Cos	کسینوس یک زاویه را محاسبه می‌کند.
CurrToStr	یک متغیر currency را به یک رشته تبدیل می‌کند.
Date	تاریخ جاری را برمی‌گرداند.

DateTimeToStr	متغیری از نوع TDateTime را به یک رشته تبدیل می‌کند.
DateToStr	متغیری از نوع TDateTime را به یک رشته تبدیل می‌کند.
Dec	یک متغیر ترتیبی را پله پله کاهش می‌دهد.
Dispose	حافظه تخصیص یافته برای یک متغیر دینامیک را آزاد می‌کند.
ExceptAddr	آدرسی را که در آن اخطار جاری تولید شده است، برمی‌گرداند.
Exit	از روال جاری خارج می‌شود.
Exp	اکسپونشیال x را محاسبه می‌کند.
FillChar	بایت‌های مجاور را با مقدار مشخصی پر می‌کند.
Finalize	یک متغیر تخصیص یافته به طور پویا را مقدار دهی اولیه نمی‌کند.
FloatToStr	یک مقدار ممیز شناور را به یک رشته تبدیل می‌کند.
FloatToStrF	با استفاده از فرمت تعیین شده، یک مقدار ممیز شناور را به یک رشته تبدیل می‌کند.
FmtLoadStr	خروجی فرمت شده را با استفاده از یک رشته فرمت منبع‌دار شده، برمی‌گرداند.
FmtStr	یک رشته فرمت شده را از یک سری آرایه سرهم بندی می‌کند.
Format	یک رشته را از یک رشته فرمت و یک سری از آرایه‌ها سرهم بندی می‌کند.
FormatDateTime	یک مقدار تاریخ و زمان را فرمت می‌کند.
FormatFloat	یک مقدار ممیز شناور را فرمت می‌کند.
FreeMem	یک متغیر دینامیک را آزاد می‌کند.
GetMem	یک متغیر دینامیک و یک اشاره‌گر به آدرس بلوک ایجاد می‌کند.
GetParentForm	فرم یا صفحه خاصیتی را که محتوی یک کنترل مشخص است، برمی‌گرداند.
Halt	پایان ناهنجار و غیر نرمال یک برنامه را راه‌اندازی می‌کند.
Hi	بایت رتبه بالای یک عبارت را به صورت یک مقدار unsigned برمی‌گرداند.
High	بالاترین مقدار واقع در دامنه یک نوع، آرایه یا رشته را برمی‌گرداند.
Inc	یک متغیر ترتیبی را به صورت پله پله نمو می‌دهد.
Initialize	یک متغیر تخصیص یافته به طور پویا را مقداردهی اولیه می‌کند.
Insert	یک زیررشته را در یک نقطه مشخص در یک رشته می‌گنجاند.
Int	بخش صحیح یک عدد حقیقی را برمی‌گرداند.
IntToStr	یک عدد صحیح را به یک رشته تبدیل می‌کند.
Length	طول یک رشته یا آرایه را برمی‌گرداند.
Lo	بایت رتبه پایین یک عبارت را به صورت یک مقدار unsigned برمی‌گرداند.
Low	پایین‌ترین مقدار واقع در دامنه یک نوع، آرایه یا رشته را برمی‌گرداند.

LowerCase	یک رشته ASCII را به حروف کوچک تبدیل می‌کند.
MaxIntValue	بزرگ‌ترین مقدار علامت‌دار واقع در یک آرایه از اعداد صحیح را برمی‌گرداند.
MaxValue	بزرگ‌ترین مقدار علامت‌دار واقع در یک آرایه را برمی‌گرداند.
MinIntValue	کوچک‌ترین مقدار علامت‌دار واقع در یک آرایه از اعداد صحیح را برمی‌گرداند.
MinValue	کوچک ترین مقدار علامت‌دار واقع در یک آرایه را برمی‌گرداند.
New	یک متغیر پویای جدید را ایجاد کرده و با یک اشاره‌گر مشخص به او ارجاع می‌کند.
Now	تاریخ و زمان جاری را برمی‌گرداند.
Ord	مقدار ترتیبی عبارتی از نوع ترتیبی را برمی‌گرداند.
Pos	اندیس اولین کاراکتر یک زیررشته مشخص واقع در یک رشته را برمی‌گرداند.
Pred	سلف (ماقبل) یک مقدار ترتیبی را برمی‌گرداند.
Ptr	آدرس مشخصی را به یک اشاره‌گر تبدیل می‌کند.
Random	اعداد تصادفی واقع در میان یک دامنه مشخصی را تولید می‌کند.
ReallocMem	متغیر پویایی را تخصیص مجدد می‌کند.
Round	مقدار یک عدد حقیقی گرد شده به نزدیک‌ترین عدد کامل را برمی‌گرداند.
SetLength	طول دینامیک یک متغیر رشته یا آرایه را تنظیم می‌کند.
SetString	محتوا و طول رشته داده شده را تنظیم می‌کند.
ShowException	یک پیغام استثناء را با آدرسش نمایش می‌دهد.
ShowMessage	یک جعبه پیغام را با یک رشته فرمت نشده و یک دکمه OK نمایش می‌دهد.
ShowMessageFmt	یک جعبه پیغام را با یک رشته فرمت شده و یک دکمه OK نمایش می‌دهد.
Sin	سینوس یک زاویه را برحسب رادیان برمی‌گرداند.
SizeOf	تعداد بایت‌های اشغال شده توسط یک متغیر یا نوع را برمی‌گرداند.
Sqr	مجذور یک عدد را برمی‌گرداند.
Sqrt	جذر یک عدد را برمی‌گرداند.
Str	یک رشته را فرمت کرده و آن را به یک متغیر برمی‌گرداند.
StrToCurr	یک رشته را به یک مقدار currency تبدیل می‌کند.
StrToDate	یک رشته را به یک فرمت تاریخ (TDateTime) تبدیل می‌کند.
StrToDateTime	یک رشته را به یک TDateTime تبدیل می‌کند.
StrToFloat	یک رشته را به یک مقدار ممیز شناور تبدیل می‌کند.
StrToInt	یک رشته را به یک عدد صحیح تبدیل می‌کند.
StrToTime	یک رشته را به یک فرمت زمان (TDateTime) تبدیل می‌کند.

StrUpper	یک رشته را در حالت حروف بزرگ برمی‌گرداند.
Succ	خلف (مابعد) یک مقدار ترتیبی را برمی‌گرداند.
Sum	مجموع عناصر یک آرایه را برمی‌گرداند.
Time	زمان جاری را برمی‌گرداند.
TimeToStr	متغیری از نوع TDateTime را به یک رشته تبدیل می‌کند.
Trunc	یک عدد حقیقی را به یک عدد صحیح کوتاه می‌کند.
UniqueString	اطمینان می‌دهد که یک رشته تنها یک ارجاع دارد. (رشته ممکن است کپی شود تا ارجاع واحدی را تولید کند.)
UpCase	یک کاراکتر را به حروف بزرگ تبدیل می‌کند.
UpperCase	یک رشته را در حالت حروف بزرگ برمی‌گرداند.
VarArrayCreate	یک آرایه واریانت ایجاد می‌کند.
VarArrayDimCount	تعداد ابعاد یک آرایه واریانت را برمی‌گرداند.
VarARrayHighBound	حد بالا را برای یک بعد واقع در یک آرایه واریانت برمی‌گرداند.
VarArrayLock	یک آرایه واریانت را قفل کرده و اشاره‌گری را به داده‌ها برمی‌گرداند.
VarArrayLowBound	حد پایین یک بعد واقع در یک آرایه واریانت را برمی‌گرداند.
VarArrayOf	یک آرایه واریانت یک بعدی را ایجاد کرده و پر می‌کند.
VarArrayRedim	یک آرایه واریانت را تغییر اندازه می‌دهد.
VarArrayRef	ارجاعی را به آرایه واریانت ارسال شده برمی‌گرداند.
VarArrayUnlock	قفل یک آرایه واریانت را باز می‌کند.
VarAsType	یک واریانت را به نوع مشخص شده تبدیل می‌کند.
VarCast	یک واریانت را با ذخیره نتیجه در یک متغیر، به یک نوع مشخص شده تبدیل می‌کند.
VarClear	یک واریانت را پاک می‌کند.
VarCopy	یک واریانت را کپی می‌کند.
VarToStr	واریانت را به رشته تبدیل می‌کند.
VarType	کد نوع واریانت مشخص را برمی‌گرداند.

فصل ۹

بسته‌ها و کتابخانه‌ها

یک کتابخانه قابل بارگذاری به طور پویا، یک کتابخانه اتصال پویا[۱] (DLL) در ویندوز یا یک فایل کتابخانه شیء اشتراکی در لینوکس است. یک کتابخانه قابل بارگذاری به طور پویا، مجموعه‌ای از روتین‌هاست که می‌توانند توسط برنامه‌ها و DLLها یا اشیاء اشتراکی دیگر فراخوانده شوند. مانند یونیت‌ها، کتابخانه‌های قابل بارگذاری به طور پویا متشکل از کد یا منابع قابل اشتراک هستند. اما این نوع از کتابخانه یک فایل اجرایی کامپایل شده به طور مجزاست که در زمان اجرا، هنگامی که برنامه از آن استفاده می‌کند، به برنامه متصل می‌شود.

برای متمایز کردن آنها از فایل‌های قابل اجرای استاندارد، در ویندوز فایل‌های محتوی DLLهای کامپایل شده با پسوند DLL. مشخص می‌شوند. در لینوکس، فایل‌های محتوی فایل‌های شیء اشتراکی با پسوند so. مشخص می‌شوند. برنامه‌های پاسکال شیئی می‌توانند DLLها یا اشیاء اشتراکی نوشته شده در زبان‌های دیگر را فراخوانی کنند و برنامه‌های نوشته شده در زبان‌های دیگر می‌توانند DLLها و اشیاء اشتراکی نوشته شده در پاسکال شیئی را فراخوانی کنند.

---

[۱] Dynamic-Link Library

فراخوانی کتابخانه‌های قابل بارگذاری به طور پویا

شما می‌توانید روتین‌های سیستم عامل را به طور مستقیم فراخوانی کنید، اما آنها تا زمان اجرای برنامه به برنامه شما متصل نمی‌شوند. این حرف به معنای این است که کتابخانه تا زمانی که شما برنامه خود را کامپایل نمایید، لازم نیست حضور داشته باشد. در ضمن به معنای این است که در آنجا هیچ ارزیابی زمان کامپایل از تلاش‌ها برای وارد کردن یک روتین وجود ندارد.

قبل از این که بتوانید از روتین‌های تعریف شده در یک شیء اشتراکی استفاده کنید، باید آنها را وارد کنید. این کار می‌تواند به دو طریق صورت گیرد: با اعلان یک روال یا تابع **external**، یا با فراخوان‌های مستقیم به سیستم عامل. جدا از این که کدام روش را استفاده کنید، روتین‌ها تا زمان اجرای برنامه، به آن متصل نمی‌شوند. پاسکال شیئی از وارد کردن متغیرها از کتابخانه‌های اشتراکی پشتیبانی نمی‌کند.

بارگذاری استاتیک

ساده‌ترین روش برای وارد کردن یک روال یا تابع اعلان آن با استفاده از راهنمای **external** است. برای مثال،

```
On Windows:   procedure DoSomething; external 'MYLIB.DLL';
On Linux:     procedure DoSomething; external 'mylib.so';
```

چنان چه این اعلان را در یک برنامه جای دهید، زمانی که برنامه راه‌اندازی شود، MYLIB.DLL (در ویندوز) یا mylib.so (در لینوکس) یک مرتبه بارگذاری می‌شود. در سراسر اجرای یک برنامه، شناسه *DoSomething* همواره به نقطه ورودی یکسانی در همان کتابخانه اشتراکی اشاره می‌کند.

اعلان‌های روتین‌های وارد شده می‌توانند به طور مستقیم در برنامه یا یونیت، جایی که آنها فراخوان می‌شوند، جای داده شوند. گرچه، برای ساده‌سازی پشتیبانی و نگه داری، شما می‌توانید اعلان‌های **external** را در یک یونیت مجزا ("import unit") جمع‌آوری نمایید که هر ثابت و نوع لازم برای میانجی‌گری با کتابخانه را نیز دربردارد. مدول‌های دیگر که از این یونیت (import) استفاده می‌کنند، می‌توانند هر روتین اعلان شده در آن را فراخوان کنند. برای آگاهی از اطلاعات بیشتر درباره اعلان‌های **external**، بخش «اعلان‌های بیرونی» را در فصل ۶ ملاحظه نمایید.

بارگذاری دینامیک

شما می‌توانید به روتین‌های یک کتابخانه از طریق فراخوان مستقیم به توابع کتابخانه‌ای OS، از جمله *LoadLibrary*، *FreeLibrary* و *GetProcAddress*، دسترسی پیدا کنید. در ویندوز، این توابع در Windows.pas اعلان شده‌اند؛ در لینوکس، آنها برای سازگاری در SysUtils.pas پیاده‌سازی شده‌اند؛ روتین‌های واقعی OS لینوکس *dlopen*، *dlclose* و *dlsym* هستند (در کایلیکس، همگی در یونیت *Libc* اعلان شده‌اند). در این حالت، از متغیرهای نوع رویه‌ای برای ارجاع به روتین‌های وارد شده استفاده کنید. برای مثال، در ویندوز یا لینوکس:

```
uses Windows, ...; {On Linux, replace Windows with SysUtils }
type
TTimeRec = record
Second: Integer;
Minute: Integer;
Hour: Integer;
end;
TGetTime = procedure(var Time: TTimeRec);
THandle = Integer;
var
Time: TTimeRec;
Handle: THandle;
GetTime: TGetTime;
...
begin
Handle := LoadLibrary('libraryname');
if Handle <> 0 then
begin
@GetTime := GetProcAddress(Handle, 'GetTime');
if @GetTime <> nil then
begin
GetTime(Time);
with Time do
WriteLn('The time is ', Hour, ':', Minute, ':', Second);
end;
FreeLibrary(Handle);
end;
end
```

چنان چه روتین‌ها را به این روش وارد کنید، تا زمانی که کد محتوی فراخوان به *LoadLibrary* اجرا نشود، کتابخانه بارگذاری نمی‌شود. کتابخانه بعداً توسط فراخوان به *FreeLibrary*، تخلیه و خارج می‌شود. این کار به شما اجازه می‌دهد تا حافظه را نگه داری کرده و برنامه خود را حتی زمانی که برخی از کتابخانه‌هایی که برنامه از آنها استفاده می‌کند، حاضر نباشند، اجرا کنید. در ضمن همان مثال می‌تواند در لینوکس به این صورت نوشته شود:

```
uses Libc, ...;
type
TTimeRec = record
```

```
Second: Integer;
Minute: Integer;
Hour: Integer;
end;
TGetTime = procedure(var Time: TTimeRec);
THandle = Pointer;
var
Time: TTimeRec;
Handle: THandle;
GetTime: TGetTime;
...
begin
Handle := dlopen('datetime.so', RTLD_LAZY);
if Handle <> 0 then
begin
@GetTime := dlsym(Handle, 'GetTime');
if @GetTime <> nil then
begin
GetTime(Time);
with Time do
WriteLn('The time is ', Hour, ':', Minute, ':', Second);
end;
dlclose(Handle);
end;
end;
```

در این حالت، هنگام وارد کردن روتین‌ها، شیء اشتراکی تا زمانی که کد محتوی فراخوان به *dlopen* اجرا نشود، بارگذاری نمی‌شود. شیء اشتراکی بعداً توسط فراخوان به *dlclose*، تخلیه و خارج می‌شود. در ضمن این کار به شما اجازه می‌دهد تا حافظه را نگه داری کرده و برنامه خود را حتی زمانی که برخی از شیء‌های اشتراکی که برنامه از آنها استفاده می‌کند، حاضر نباشند، اجرا کنید.

نوشتن کتابخانه‌های قابل بارگذاری به طور پویا

منبع اصلی مربوط به یک کتابخانه قابل بارگذاری به طور پویا همانند منبع اصلی مربوط به یک برنامه است، به جز این که به جای برنامه که با **program** شروع می‌شود، با واژه کلیدی **library** آغاز می‌شود.

تنها روتین‌هایی که یک کتابخانه به طور ضمنی صادر می‌کند، به منظور وارد کردن توسط کتابخانه‌ها یا برنامه‌های دیگر، در دسترس هستند. مثال زیر یک کتابخانه با دو تابع صادرشده، *Min* و *Max* را نشان می‌دهد.

```
library MinMax;
function Min(X, Y: Integer): Integer; stdcall;
begin
if X < Y then Min := X else Min := Y;
end;
function Max(X, Y: Integer): Integer; stdcall;
begin
```

```
if X > Y then Max := X else Max := Y;
end;
exports
Min,
Max;
begin
end.
```

چنان چه بخواهید کتابخانه شما برای برنامه‌های نوشته شده در زبان‌های دیگر در دسترس باشد، مطمئن‌تر این است که **stdcall** را در اعلان توابع صادر شده تصریح نمایید. زبان‌های دیگر ممکن است از قرارداد فراخوانی پیش فرض **register** پاسکال شیئی پشتیبانی نکنند.

کتابخانه‌ها می‌توانند از یونیت‌های متعددی ایجاد شوند. در این حالت، خیلی اوقات فایل منبع کتابخانه به یک شرط **uses**، یک شرط **exports** و کد مقداردهی اولیه تقلیل پیدا می‌کند. برای مثال،

```
library Editors;
uses EdInit, EdInOut, EdFormat, EdPrint;
exports
InitEditors,
DoneEditors name Done,
InsertText name Insert,
DeleteSelection name Delete,
FormatSelection,
PrintSelection name Print,
...
SetErrorHandler;
begin
InitLibrary;
end.
```

شما می‌توانید شروط **exports** را در بخش **interface** یا **implementation** یک یونیت قرار دهید. هر کتابخانه‌ای که یک چنین یونیتی را در شرط **uses** خودش جای دهد، به طور خودکار روتین‌های فهرست شده در شرط **exports** یونیت را صادر می‌کند— بدون نیاز به یک شرط **exports** که مال خودش باشد.

راهنمای **local**، که روتین‌ها را برای صدور غیر قابل دسترس می‌سازد، خاص-پلت‌فرم بوده و تأثیری در برنامه‌نویسی ویندوز ندارد. در لینوکس، راهنمای **local** یک بهینه‌سازی عملکرد ناچیز برای روتین‌هایی که به درون یک کتابخانه کامپایل شده‌اند اما صادر نشده‌اند، فراهم می‌کند. این راهنما می‌تواند برای روال‌ها و توابع استاندارد تعیین شود، اما برای متدها نه. یک روتین اعلان شده با **local** — برای مثال،

```
function Contraband(I: Integer): Integer; local;
```

— ثبات EBX را تجدید نمی‌کند و از این رو

- نمی‌تواند از یک کتابخانه صادر شود.
- نمی‌تواند در بخش **interface** یک یونیت اعلان شود.
- نمی‌تواند آدرسش را از یک متغیر نوع رویه‌ای گرفته باشد یا به آن تخصیص داده باشد.
- اگر یک روتین اسمبلر خالص باشد، نمی‌تواند از یونیت دیگری فراخوانده شود مگر این که فراخواننده EBX را تنظیم کرده باشد.

شرط exports

چنان چه روتینی در یک شرط **exports** لیست شده باشد، صادر می‌شود. شرط **exports** قالب زیر را دارد

```
exports entry1, ..., entryn;
```

جایی که *entry* از اسم یک روال، تابع یا متغیر (که باید قبل از شرط **exports** اعلان شده باشد) تشکیل می‌شود، که توسط یک لیست پارامتر (تنها اگر در حال صدور یک روتین که سربارگذاری شده است، باشیم) و یک تصریح کننده **name** اختیاری پی گرفته می‌شود. شما می‌توانید اسم تابع یا روال را با اسم یک یونیت قیددار کنید. (در ضمن *entry*ها می‌توانند حاوی راهنمای **resident** باشند، که برای سازگاری با گذشته پشتیبانی می‌شود و توسط کامپایلر نادیده گرفته می‌شود.)

صرفاً در ویندوز، یک تصریح کننده **index** از راهنمای **index** که با یک ثابت عددی در فاصله 1 و 2,147,483,647 پی گرفته می‌شود، تشکیل می‌شود. (برای برنامه‌های کارآمدتر، از مقادیر پایین اندیس استفاده کنید.) اگر یک مدخل هیچ تصریح کننده **index** نداشته باشد، برای روتین، به طور خودکار یک عدد در جدول صدور تخصیص داده می‌شود.

توجه استفاده از تصریح کننده‌های **index**، که تنها برای سازگاری با گذشته پشتیبانی می‌شوند، دلسرد کننده می‌باشد و ممکن است سبب بروز مشکلاتی برای ابزارهای طراحی و توسعه دیگر شود.

تصریح کننده **name** از راهنمای **name** که با یک ثابت رشته‌ای پی گرفته می‌شود، تشکیل می‌شود. اگر یک مدخل هیچ تصریح کننده **name** نداشته باشد، روتین تحت نام اعلان شده اصلی‌اش، با همان

املا و حالت حروف، صادر می‌شود. زمانی که می‌خواهید یک روتین را تحت یک نام متفاوت صادر کنید، از یک شرط **name** استفاده کنید. برای مثال،

```
exports
DoSomethingABC name 'DoSomething';
```

هرگاه یک تابع یا روال سربارگذاری شده را از یک کتابخانه قابل بارگذاری به طور پویا صادر کردید، باید لیست پارامترهایش را در شرط **exports** تعیین کنید. برای مثال،

```
exports
Divide(X, Y: Integer) name 'Divide_Ints',
Divide(X, Y: Real) name 'Divide_Reals';
```

در ویندوز، تصریح کننده‌های **index** را در مدخل‌های مربوط به روتین‌های سربارگذاری شده، قرار ندهید.

شرط **exports** می‌تواند در هر جایی و به هر تعداد بخش اعلان یک برنامه یا کتابخانه، یا در بخش **interface** یا **implementation** یک یونیت ظاهر شود. برنامه‌ها به ندرت شامل یک شرط **exports** هستند.

کد مقداردهی اولیه کتابخانه

دستورات واقع در بلوک یک کتابخانه *کد مقداردهی اولیه* کتابخانه را تشکیل می‌دهند. این دستورات، هرگاه که کتابخانه بارگذاری شود، یک مرتبه اجرا می‌شوند. آنها نوعاً وظایفی مانند ثبت کردن کلاس‌های ویندوز و مقداردهی اولیه متغیرها را انجام می‌دهند. در ضمن کد مقداردهی اولیه کتابخانه می‌تواند یک روال خروج را با استفاده از متغیر *ExitProc* نصب کند (بخش «روال‌های خروج» را در فصل۱۲ ملاحظه نمایید)؛ زمانی که کتابخانه تخلیه و خارج گردد روال خروج اجرا می‌شود.

با نشاندن متغیر *ExitCode* به یک مقدار غیر صفرکد مقداردهی اولیه کتابخانه یک خطا مخابره می‌کند. *ExitCode* در یونیت *System* اعلان شده است و مقدار پیش فرضش صفر است که این مقدار نشانگر مقداردهی موفقیت‌آمیز است. چنان چه کد مقداردهی یک کتابخانه *ExitCode* را برابر مقدار دیگری قرار دهد، کتابخانه تخلیه و خارج شده و فراخواننده برنامه از یک عدم موفقیت آگاه می‌شود. به طور مشابه، اگر یک استثنای اداره نشده در طی اجرای کد مقداردهی اولیه اتفاق بیفتد، فراخواننده برنامه از یک عدم موفقیت برای بارگذاری کتابخانه آگاه می‌شود.

در اینجا یک مثال برای یک کتابخانه با کد مقداردهی و یک روال خروج آورده شده است.

```
library Test;
var
SaveExit: Pointer;
procedure LibExit;
begin
...                         // library exit code
ExitProc := SaveExit;       // restore exit procedure chain
end;
begin
...                         // library initialization code
SaveExit := ExitProc;       // save exit procedure chain
ExitProc := @LibExit;       // install LibExit exit procedure
end.
```

زمانی که یک کتابخانه تخلیه و خارج می‌شود، روال‌های خروجش توسط فراخوان‌های مکرر به آدرس ذخیره شده در *ExitProc* اجرا می‌شوند، تا این که *ExitProc* برابر **nil** شود. بخش‌های مقداردهی (initialization) همه یونیت‌های استفاده شده توسط یک کتابخانه قبل از کد مقداردهی (initialization) یک کتابخانه اجرا می‌شوند و بخش‌های اتمام (finalization) این یونیت‌ها بعد از روال خروج کتابخانه اجرا می‌شوند.

متغیرهای سراسری در یک کتابخانه

متغیرهای سراسری اعلان شده در یک کتابخانه اشتراکی نمی‌توانند توسط یک برنامه پاسکال شیئی وارد شوند. یک کتابخانه می‌تواند هم زمان توسط برنامه‌های متعددی به کار برده شود، اما هر برنامه یک کپی از کتابخانه را در فضای پردازش خود با مجموعه‌ای از متغیرهای سراسری شخصی مال خود دارد. برای کتابخانه‌های متعدد— یا وهله‌های متعددی از یک کتابخانه— برای تسهیم حافظه، آنها باید از فایلهای نگاشته شده حافظه[۱] استفاده کنند.

کتابخانه‌ها و متغیرهای سیستم

متغیرهای متعدد که در یونیت *System* اعلان شده‌اند، در برنامه نویسی کتابخانه‌ها مورد توجه ویژه‌ای هستند. برای این که مشخص کنید که آیا کد در حال اجرا در یک برنامه است یا در یک کتابخانه، از *IsLibrary* استفاده کنید؛ *IsLibrary* همواره در برنامه *False* و در یک کتابخانه *True* است؛ در طی دوره

---

[۱] Memory-mapped

زندگی یک کتابخانه، *HInstance* محتوی دستگیره وهله آن است. *CmdLine* همواره در یک کتابخانه برابر **nil** است.

متغیر *DLLProc* به یک کتابخانه اجازه می‌دهد تا فراخوان‌هایی را که سیستم عامل برای نقطه مدخل کتابخانه انجام می‌دهد، رصد کند. این مشخصه معمولاً تنها برای کتابخانه‌هایی که از چندریسمانی پشتیبانی می‌کنند، به کار برده می‌شود. *DLLProc* هم در ویندوز و هم در لینوکس در دسترس است اما کاربرد آن روی هرکدام از آنها فرق می‌کند. در ویندوز، *DLLProc* در برنامه‌های چندریسمانی به کار برده می‌شود؛ در لینوکس، برای تعیین این که چه زمانی کتابخانه شما تخلیه وخارج خواهد شد، به کار برده می‌شود. شما بایستی به جای روال‌های خروج، از بخش‌های اتمام (finalization)، برای تمامی وضعیت‌های خروج استفاده کنید. (بخش «بخش finalization» را در فصل۳ ملاحظه نمایید.)

برای رصد فراخوان‌های سیستم عامل، یک روال callback ایجاد کنید که پارامتر صحیح واحدی می‌گیرد— برای مثال،

```
procedure DLLHandler(Reason: Integer);
```

— و آدرس روال را به متغیر *DLLProc* تخصیص دهید. زمانی که روال فراخوانده می‌شود، یکی از مقادیر زیر را به آن ارسال می‌کند.

DLL_PROCESS_DETACH	نشان می‌دهد که کتابخانه از فضای آدرس فرایند فراخوانی به صورت نتیجه‌ای از یک خروج بی نقص یا یک فراخوان به *FreeLibrary* یا (*dlclose* در لینوکس)، تفکیک می‌شود.
DLL_THREAD_ATTACH	نشان می‌دهد که فرایند جاری در حال ایجاد یک ریسمان جدید (تنها در ویندوز) است.
DLL_THREAD_DETACH	نشان می‌دهد که یک ریسمان درحال خروج به روش بی نقصی (تنها در ویندوز) است.

در لینوکس، این مقادیر در یونیت Libc تعریف شده‌اند.

در بدنه روال، بسته به اینکه کدام پارامتر به روال ارسال شده باشد، شما می‌توانید عملیات‌هایی را برای اتخاذ کردن، مشخص نمایید.

استثناها و خطاهای زمان اجرا در کتابخانه‌ها

هرگاه در یک کتابخانه قابل بارگذاری به طور پویا یک استثناء صادر شده اما اداره نشود، این استثناء به بیرون از کتابخانه و به فراخواننده منتشر می‌شود. اگر فراخوانی برنامه یا کتابخانه خود در پاسکال شیئی نوشته شده باشد، استثناء می‌تواند از طریق یک دستور نرمال **try...except** اداره شود.

توجه تحت لینوکس این امر صرفاً زمانی امکان پذیر خواهد بود که کتابخانه و برنامه هر دو با مجموعه یکسانی از بسته‌های زمان اجرا (که حاوی کد EH است) ایجاد شده باشند یا اگر هر دو متصل به ShareExcept باشند.

اگر برنامه فراخواننده یا کتابخانه در زبان دیگری نوشته شده باشد، استثناء می‌تواند به صورت یک استثنای سیستم عامل با کد استثنای $0EEDFACE اداره شود. اولین مدخل واقع در آرایه *ExceptionInformation* از رکورد استثنای سیستم عامل حاوی آدرس استثناست و دومین مدخل حاوی یک ارجاع به شیء استثنای پاسکال شیئی است.

به طور کلی، شما نبایست اجازه دهید تا استثناها از کتابخانه شما بگریزند. در ویندوز، استثناهای دلفی به مدل اخطار OS نگاشته می‌شوند؛ لینوکس هیچ مدل اخطاری ندارد.

اگر یک کتابخانه از یونیت *SysUtils* استفاده نکند، پشتیبان استثناء غیر فعال می‌شود. در این حالت، هر گاه یک خطای زمان اجرا در کتابخانه رخ دهد، برنامه فراخواننده خاتمه می‌یابد. از آن جایی که کتابخانه هیچ راهی ندارد که بداند آیا از یک برنامه پاسکال شیئی فراخوانده شده بود یا نه، نمی‌تواند روال‌های خروج برنامه را احضار کند؛ برنامه به سادگی متوقف شده و از حافظه حذف می‌شود.

مدیر حافظه تسهیم شده (تنها برای ویندوز)

در ویندوز، اگر یک DLL روتین‌هایی را صادر کند که رشته‌های بلند یا آرایه‌های دینامیک را به عنوان پارامتر یا نتایج توابع ارسال می‌کنند (خواه به طور مستقیم یا خواه به طور آشیانه‌ای در رکوردها یا

اشیاء)، در این صورت DLL و برنامه‌های (یا DLLهای) مشتری آن همگی باید از یونیت *ShareMem* استفاده کنند. به طور مشابه، چنان چه یک برنامه یا DLL حافظه را با *New* یا *GetMem* تخصیص دهد که این حافظه با فراخوان به *Dispose* یا *FreeMem* در مدول دیگر آزاد می‌شود، مطلب گفته شده قبلی صحیح خواهد بود. *ShareMem* همواره بایستی اولین یونیت لیست شده در شرط **uses** هر برنامه یا کتابخانه، جایی که اتفاق می‌افتد، باشد.

ShareMem یونیت واسط برای مدیر حافظه BORLANDMM.DLL است، که به مدول‌ها اجازه می‌دهد تا حافظه تخصیص یافته به طور پویا را تسهیم کنند. BORLANDMM.DLL باید با برنامه‌ها و DLLهایی که از *ShareMem* استفاده می‌کنند منتشر شود. هرگاه یک برنامه یا DLL از *ShareMem* استفاده کند، مدیر حافظه آن با مدیر حافظه واقع در BORLANDMM.DLL جایگزین می‌شود. لینوکس از تابع *malloc* که در glibc تعریف شده است، برای مدیریت حافظه تسهیم یافته استفاده می‌کند.

بسته‌ها

یک بسته[۱] کتابخانه کامپایل شده به خصوصی است که توسط برنامه‌ها، IDE، یا هر دو به کار برده می‌شود. زمانی که کد بدون تأثیر بر کد منبع مسقر می‌شود، بسته‌ها به شما اجازه بازآرایی می‌دهند. این امر برخی اوقات منتسب به عنوان *افراز برنامه*[۲] است.

زمانی که یک کاربر برنامه‌ای را اجرا می‌کند، بسته‌های زمان اجرا عاملیت و کارکردی را عرضه می‌کنند. بسته‌های زمان طراحی به منظور نصب اجزاء در IDE و ایجاد ویرایشگرهای خاصیت ویژه برای اجزاء سفارشی به کار برده می‌شوند. یک بسته واحد می‌تواند در هر دو زمان طراحی و اجرا عمل کند و خیلی اوقات بسته‌های زمان طراحی با ارجاع دادن به بسته‌های زمان اجرا در شروط **requires**شان عمل می‌کنند.

برای متمایز کردن بسته‌ها از کتابخانه‌های دیگر، بسته‌ها در فایل‌ها ذخیره می‌شوند:

- در ویندوز، فایل‌های بسته با پسوند bpl (Borland package library). خاتمه می‌یابند.
- در لینوکس، بسته‌ها معمولاً با پیشوند bpl شروع می‌شوند و یک پسوند so. دارند.

---

[۱] Package

[۲] Application partitioning

معمولاً، بسته‌ها زمانی که یک برنامه شروع می‌شود، به طور استاتیک بارگذاری می‌شوند. اما شما می‌توانید روتین‌های *LoadPackage* و *UnloadPackage* (تعریف شده در یونیت *SysUtils*) را برای بارگذاری بسته‌ها به طور دینامیک به کار برید.

توجه هنگامی که یک برنامه بسته‌ها را مورد استفاده قرار می‌دهد، اسم هر یونیت بسته‌ای هنوز هم بایستی در شرط **uses** هر فایل منبعی که به آن ارجاع می‌کند، ظاهر شود.

اعلان بسته‌ها و فایل‌های منبع

هر بسته در یک فایل منبع مجزا اعلان می‌شود، که باید با پسوند dpk. ذخیره شود تا از اشتباه گرفته شدن با فایل‌های دیگری که محتوی کد پاسکال شیئی هستند، اجتناب شود. یک فایل منبع بسته شامل اعلان‌های نوع، داده، روال یا تابع نیست. در عوض، این فایل حاوی موارد زیر است

- یک اسم برای بسته.
- لیستی از بسته‌های دیگر موردنیاز بسته جدید. اینها بسته‌هایی هستند که بسته جدید به آنها متصل می‌شود.
- لیستی از فایل‌های یونیت دربرگرفته شده توسط، یا محدود در بسته، برای زمانی که بسته کامپایل می‌شود. بسته در اصل یک لفافه برای این یونیت‌های کد منبع است، که عاملیت و کارکرد بسته کامپایل شده را فراهم می‌کنند.

اعلان یک بسته به صورت زیر است

```
package packageName;
requiresClause;
containsClause;
end.
```

جایی که *packageName* هر گونه شناسه معتبری است. *requiresClause* و *containsClause* هر دو اختیاری هستند. برای مثال، کد زیر بسته *DATAX* را اعلان می‌کند.

```
package DATAX;
requires
baseclx,
visualclx;
contains Db, DBLocal, DBXpress, ... ;
end.
```

شرط **requires** دیگر بسته‌های بیرونی استفاده شده توسط بسته اعلان شده را لیست می‌کند. این شرط از راهنمای **requires** که توسط لیستی از اسامی بسته که توسط ویرگول از هم جدا می‌شوند و یک نقطه ویرگول پی گرفته می‌شود، تشکیل می‌شود. اگر یک بسته به بسته‌های دیگر ارجاع نکند، به شرط **requires** نیازی ندارد.

شرط **contains** مشخص می‌کند که فایل‌های یونیت کامپایل شده و به میان بسته محدود شوند. این شرط از راهنمای **contains**، که توسط لیستی از اسامی بسته که با ویرگول از هم جدا می‌شوند، و یک نقطه ویرگول پی گرفته می‌شود، تشکیل می‌شود. هر اسم یونیت می‌تواند با واژه کلیدی **in** و اسم یک فایل منبع، با یا بدون یک مسیر دایرکتوری، در میان علائم نقل قول منفرد ' '، پی گرفته شود؛ در ضمن مسیرهای دایرکتوری می‌توانند مطلق یا نسبی باشند. برای مثال،

```
contains MyUnit in 'C:\MyProject\MyUnit.pas'; // Windows
contains MyUnit in '\home\developer\MyProject\MyUnit.pas'; // Linux
```

توجه متغیرهای ریسمان–محلی (اعلان شده با **threadvar**) در یک یونیت بسته‌ای نمی‌توانند از مشتری‌هایی که از بسته استفاده می‌کنند، قابل دسترسی باشند.

نام گذاری بسته‌ها

یک بسته کامپایل شده با فایل‌های تولید شده متعددی درگیر است. برای مثال، فایل منبع مربوط به بسته‌ای به نام *DATAX*، فایل DATAX.dpk است، که از آن کامپایلر یک فایل قابل اجرا و یک تصویر باینری به نام‌های زیر تولید می‌کند

- در ویندوز : DATAX.bpl و DATAX.dcp
- در لینوکس : bplDATAX.so و DATAX.dcp

DATAX برای اشاره به بسته واقع در شروط **requires** بسته‌های دیگر یا هنگام استفاده از بسته در یک برنامه، به کار برده می‌شود. اسامی بسته بایستی در میان یک پروژه منحصر به فرد باشند.

شرط requires

شرط **requires** بسته‌های بیرونی دیگری را که توسط بسته جاری به کار برده شده‌اند، لیست می‌کند. این شرط مانند شرط **uses** در یک فایل یونیت عمل می‌کند. یک بسته بیرونی لیست شده در شرط

requires در زمان کامپایل به طور خودکار به هر برنامه‌ای که هم از بسته جاری و هم یکی از یونیت‌های دربرگرفته شده در بسته بیرونی استفاده می‌کند، متصل می‌شود.

اگر فایل‌های یونیت دربرگرفته شده در یک بسته ارجاعاتی به یونیت‌های بسته‌ای دیگر انجام دهند، بسته‌های دیگر بایستی در شرط **requires** اولین بسته قرار گیرند. اگر بسته‌های دیگر از شرط **requires** حذف شوند، کامپایلر یونیت‌های ارجاع شده را از فایل‌های dcu. (ویندوز) یا dpu. (لینوکس) آنها بارگذاری می‌کند.

پرهیز از ارجاعات چرخشی به بسته‌ها

بسته‌ها نمی‌توانند حاوی ارجاعات چرخشی در شروط **requires** خود باشند. این سخن به این معناست که

- یک بسته نمی‌تواند به خودش در شرط **requires** متعلق به خودش ارجاع کند.
- یک زنجیره از ارجاعات باید بدون ارجاع کردن به هر بسته در زنجیره ارجاعات خاتمه یابد. اگر بسته A، بسته B را لازم داشته باشد، در این صورت بسته B نمی‌تواند نیازمند بسته A باشد؛ اگر بسته A، بسته B را لازم داشته باشد و بسته B، بسته C را لازم داشته باشد، در این صورت بسته C نمی‌تواند نیازمند بسته A باشد.

ارجاعات بسته تکراری

کامپایلر از ارجاعات تکراری در شرط **requires** یک بسته صرفنظر می‌کند. گرچه، به منظور وضوح و خوانایی برنامه‌نویسی، ارجاعات تکراری بایست حذف شوند.

شرط contains

شرط **contains** مشخص می‌کند که فایل‌های یونیت مقید به بسته‌ها باشند. پسوندهای اسم فایل را در شرط **contains** جای ندهید.

پرهیز از کد منبع زائد uses

یک بسته نمی‌تواند در شرط **contains** بسته دیگر یا شرط **uses** یک یونیت لیست شود. همه یونیت‌هایی که به طور مستقیم در شرط **contains** یک بسته جای گرفته‌اند، یا به طور غیرمستقیم در

شروط **uses** آن یونیت‌ها قرار گرفته‌اند، در زمان کامپایل به میان بسته مقید می‌شوند. یونیت‌های جای گرفته در بسته (چه به طور مستقیم و چه به طور غیر مستقیم) نمی‌توانند در هر بسته‌ای که در شرط **requires** آن بسته مورد ارجاع واقع شده، جای داده شوند. یک یونیت نمی‌تواند در بیشتر از یک یونیت استفاده شده توسط همان برنامه، (به طور مستقیم یا غیر مستقیم) جای داده شود.

کامپایل بسته‌ها

بسته‌ها معمولاً از طریق IDE با استفاده از فایل‌های dpk. تولید شده توسط ویرایشگر بسته، کامپایل می‌شوند. در ضمن شما می‌توانید فایل‌های dpk. را مستقیماً از خط فرمان کامپایل کنید. هرگاه پروژه‌ای را که دارای یک بسته است بنا می‌کنید، چنان چه لازم باشد، بسته به طور ضمنی کامپایل مجدد می‌شود.

فایل‌های تولید شده

جدول زیر فایل‌های تولید شده توسط کامپایل موفق یک بسته را فهرست‌بندی می‌کند.

Table 9.1 فایل‌های بسته کامپایل شده

پسوند	مضمون
dcp	یک تصویر باینری محتوی یک هدر بسته و الحاق تمامی فایل‌های dcu (Windows) یا dpu (Linux) در یک بسته. یک فایل مجزای dcp برای هر بسته ایجاد می‌شود. اسم پایه برای dcp، اسم پایه فایل منبع dpk است.
dcu (Windows) dpu (Linux)	یک تصویر باینری مربوط به یک فایل یونیت قرار گرفته در یک بسته. هرگاه که لازم باشد یک فایل dcu یا dpu به ازای هر فایل یونیت ایجاد می‌شود.
.bpl on Windows bpl<package>.so on Linux	بسته زمان اجرا. این فایل یک کتابخانه اشتراکی با مشخصه‌های به خصوص بورلند است. اسم پایه برای بسته، اسم پایه فایل منبع dpk است.

راهنماهای کامپایلر و سوییچ‌های خط فرمان متعددی برای کامپایل بسته در دسترس هستند.

راهنماهای کامپایلر ویژه بسته

جدول زیر راهنماهای ویژه بسته را فهرست می‌کند؛ این راهنماها می‌توانند در کد منبع درج شود.

Table 9.2 راهنماهای کامپایلر ویژه بسته (**Package**)

راهنما	منظور
{$IMPLICITBUILD OFF}	بعداً از کامپایل شدن مجدد یک بسته به طور ضمنی جلوگیری می‌کند. هنگام کامپایل بسته‌هایی که عاملیت ترازپایین را فراهم می‌کنند، که به ندرت در میان ساخت‌ها تغییر می‌کنند یا کد منبع آنها توزیع نخواهد شد، از این راهنما در فایل‌های dpk. استفاده کنید.
{$G–} or {$IMPORTEDDATA OFF}	ایجاد ارجاعات داده وارد شده را غیر فعال می‌کند. این دستور بازده دسترسی حافظه را افزایش می‌دهد، اما یونیت را، جایی که اتفاق می‌افتد، از ارجاع کردن به متغیرهایی واقع در بسته‌های دیگر بازمی‌دارد.
{$WEAKPACKAGEUNIT ON}	یونیت را به طور ضعیف (''weakly'') بسته بندی می‌کند.
{$DENYPACKAGEUNIT ON}	یونیت را از جای گرفتن در یک بسته بازمی‌دارد.
{$DESIGNONLY ON}	یونیت را به منظور استقرار در IDE کامپایل می‌کند. (در فایل dpk. قرار می‌دهد.)
{$RUNONLY ON}	بسته‌ها را صرفاً به صورت زمان اجرا کامپایل می‌کند. (در فایل dpk. قرار می‌دهد.)

جای دادن **{$DENYPACKAGEUNIT ON}** در کد منبع یونیت را از بسته‌ای شدن بازمی‌دارد. جای دادن **{$G–}** یا **{$IMPORTEDDATA OFF}** می‌تواند یک بسته را از استفاده شدن در همان برنامه با بسته‌های دیگر باز دارد. راهنماهای کامپایلر دیگر، چنان چه مقتضی باشد، می‌توانند در کد منبع بسته جای گیرند.

سوییچ‌های کامپایلر خط فرمان ویژه بسته

سوییچ‌های ویژه بسته زیر برای کامپایلر خط فرمان در دسترس هستند. برای جزییات بیشتر راهنمای درون خطی را ملاحظه نمایید.

Table 9.3 سوییچ‌های کامپایلر خط فرمان ویژه بسته (**Package**)

سوئیچ	منظور
–G$–	ایجاد ارجاعات داده وارد شده را غیرفعال می‌کند. استفاده از این سوئیچ بازده دسترسی حافظه را افزایش می‌دهد، اما بسته‌های کامپایل شده با آن را از ارجاع کردن به متغیرهای واقع در بسته‌های دیگر بازمی‌دارد.
–LE *path*	دایرکتوریی را که فایل بسته کامپایل شده در آن جای خواهد گرفت، مشخص می‌کند.
–LN *path*	دایرکتوریی را که فایل dcp بسته در آن جای خواهد گرفت، مشخص می‌کند.
–LU *packageName [...;packageName2;]*	بسته‌های زمان اجرای اضافی را به منظور استفاده در یک برنامه تعیین می‌کند. هنگام کامپایل یک پروژه استفاده می‌شود.
–Z	در آینده از کامپایل شدن مجدد یک بسته به طور ضمنی جلوگیری می‌کند. هنگام کامپایل بسته‌هایی که عاملیت ترازپایین را فراهم می‌کنند، که به ندرت در میان buildها تغییر می‌کنند یا کد منبع آنها توزیع نخواهد شد، از این سوییچ استفاده کنید.

استفاده از سوئیچ **–$G–** می‌تواند یک بسته را از استفاده شدن در همان برنامه با بسته‌های دیگر بازدارد. چنان چه مقتضی باشد، گزینه‌های دیگر خط فرمان می‌توانند هنگام کامپایل بسته‌ها، مورد استفاده قرار گیرند.

فصل ۱۰

واسط‌های شیء

یک *واسط شیء*[۱] — یا در واقع *واسط* (*interface*)— متدهایی را تعریف می‌کند که می‌توانند توسط یک کلاس پیاده‌سازی شوند. واسط‌ها مانند کلاس‌ها اعلان می‌شوند، اما نمی‌توانند به طور مستقیم نمونه‌سازی شده و تعاریف متد مال خود را ندارند. ترجیحاً، این وظیفه هر کلاسی است که یک واسط را پشتیبانی کند تا پیاده‌سازی‌هایی را برای متدهای واسط آماده کند. متغیری از یک نوع واسط می‌تواند به یک شیء ارجاع کند که کلاس این شیء آن واسط را پیاده‌سازی می‌کند؛ گرچه، تنها متدهای اعلان شده در واسط می‌توانند با استفاده از همچو متغیری فراخوان شوند.

واسط‌ها برخی مزایای توارث چندگانه را بدون مشکلات معناشناختی، عرضه می‌کنند. در ضمن آنها برای استفاده از مدل‌های شیء توزیع شده ضروری هستند. اشیاء سفارشی ساخته شده که از واسط‌ها پشتیبانی می‌کنند، می‌توانند با اشیاء نوشته شده در C++، Java و زبان‌های دیگر فعل و انفعال متقابل داشته باشند.

---

[۱] Object interface

نوع‌های واسط

واسط‌ها، مانند کلاس‌ها، تنها می‌توانند در بیرونی‌ترین دامنه یک برنامه یا یونیت و نـه در میـان اعـلان یک تابع یا روال، اعلان شوند. اعلان یک نوع واسط قالب زیر را دارد

```
type interfaceName = interface (ancestorInterface)
['{GUID}']
memberList
end;
```

جایی که (*ancestorInterface*) و ['{*GUID*}'] اختیاری هستند. در اغلب ملاحظات، اعلان‌هـای واسـط مشابه اعلان‌های کلاس به نظر می‌رسند، اما محدودیت‌های زیر به آنها اعمال می‌شوند.

- *memberList* تنها می‌تواند حاوی متدها و خاصیت‌ها باشد. فیلدها در واسط‌ها مجاز نیستند.
- از آن جایی که یک واسط هیچ فیلدی ندارد، تصریح کننده‌های **read** و **write** خاصـیت بایـد متد باشند.
- همـه اعضـای یـک واسـط عمـومی(public) هسـتند. تصـریح کننـده‌هـای میـدان دیـد و تصریح‌کننده‌های انباره در واسط‌ها مجاز نیستند. (اما یک خاصیت آرایه می‌تواند بـه صـورت **default** اعلان شود.)
- واسط‌ها هیچ سازنده یا تخریب کننده‌ای ندارند. آنها نمی‌توانند نمونه‌سازی شـوند، بـه جـز از طریق کلاس‌هایی که متدهای آنها را پیاده‌سازی می‌کنند.
- متدها نمی‌توانند به صورت **virtual**، **dynamic**، **abstract** یا **override** اعلان شـوند. از آن جایی که واسط‌ها نمی‌توانند متدهای مال خود را پیاده‌سازی کنند، این نقش‌ها معنایی ندارند.

در اینجا مثالی از اعلان یک واسط آورده شده است:

```
type
IMalloc = interface(IInterface)
['{00000002-0000-0000-C000-000000000046}']
function Alloc(Size: Integer): Pointer; stdcall;
function Realloc(P: Pointer; Size: Integer): Pointer; stdcall;
procedure Free(P: Pointer); stdcall;
function GetSize(P: Pointer): Integer; stdcall;
function DidAlloc(P: Pointer): Integer; stdcall;
procedure HeapMinimize; stdcall;
end;
```

در برخی اعلان‌های واسط، واژه کلیدی **interface** توسط **dispinterface** جـایگزین مـی‌شـود. ایـن ساختار (همراه با فرمان‌های **dispid**، **readonly** و **writeonly**) ویژه–پلت‌فرم بوده و در برنامه‌نویسی لینوکس به کار برده نمی‌شود.

IInterface و وراثت

یک واسط، مانند یک کلاس، تمامی متدهای نیای خود را بـه ارث مـی‌بـرد. امـا واسـط‌هـا، بـرخلاف کلاس‌ها، متدها را **پیاده‌سازی نمی‌کنند**. چیزی که واسط به ارث می‌برد وظیفه پیاده‌سازی متدهاست—وظیفه‌ای که به هر کلاسی که واسط را پشتیبانی کند محول می‌شود.

اعلان یک واسط می‌تواند یک واسط نیا را مشخص کند. چنان چه هیچ نیایی تعیین نشـود، واسـط بـه طور مستقیم از *IInterface* مشتق می‌شود، که این واسط پایه‌ای در یونیت *System* تعریف شده اسـت و نیای نهایی تمامی واسط‌های دیگر اسـت. *IInterface* سـه متـد را تعریـف مـی‌کنـد: *QueryInterface*، *_AddRef* و *_Release*.

توجـه *IInterface* معـادل بـا *IUnknown* اسـت. شـما بایسـتی بـه طـور کلـی از *IInterface* برای برنامه‌های مستقل از پلت‌فرم استفاده کنیـد و اسـتفاده از *IUnknown* را برای برنامه‌های خاصی که حاوی متعلقات ویندوز هستند، بگذارید.

QueryInterface اسبابی را برای حرکت آزادانه در میان واسط‌های مختلفـی کـه یـک شـیء پشـتیبانی می‌کند، فراهم می‌کند. *_AddRef* و *_Release* مدیریت حافظه دوره حیات[1] را بـرای ارجاعـات واسـط فراهم می‌کنند. آسان‌ترین روش برای پیاده‌سازی این متـدها ایـن اسـت کـه کـلاس پیـاده‌سـازی را از *TInterfacedObject* متعلق به یونیت *System*، استنتاج کرد. در ضمن با پیـاده‌سـازی هـر یـک از ایـن متدها به صورت یک تابع تهی می‌توان از دست آنها خـلاص شـد؛ گرچـه، اشـیای COM (تنهـا بـرای ویندوز) باید از طریق *_AddRef* و *_Release* مدیریت شوند.

[1] Lifetime Memory Management

شناسایی واسط

اعلان یک واسط می‌تواند یک شناسه سراسری منحصر به فرد (GUID) تعیین کند، که توسط یک لیترال رشته محصور در میان **[]** که بلافاصله قبل از لیست اعضاء می‌آید، نمایش داده می‌شود. بخش GUID اعلان باید قالب زیر را داشته باشد

```
['{xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx}']
```

جایی که x یک رقم هگزادسیمال (از ۰ تا ۹ یا A تا F — ارقام مبنای شانزده) می‌باشد. در ویندوز، ویرایشگر کتابخانه نوع به طور خودکار GUIDها را برای واسط‌های جدید تولید می‌کند؛ در ضمن شما می‌توانید با فشار دادن *Ctrl+Shift+G* در ویرایشگر کد، GUIDها را تولید کنید (در لینوکس، باید از *Ctrl+Shift+G* استفاده کنید).

یک GUID یک مقدار باینری ۱۶ بایت است که به طور منحصر به فردی یک واسط را مشخص می‌کند. اگر یک واسط یک GUID داشته باشد، شما می‌توانید از جستار واسط برای گرفتن ارجاعات به پیاده‌سازی‌اش استفاده کنید. (بخش «جستار واسط» را در همین فصل ملاحظه نمایید.) نوع‌های *TGUID* و *PGUID*، که در یونیت *System* اعلان شده‌اند، برای دست کاری GUIDها به کار برده می‌شوند.

```
type
PGUID = ^TGUID;
TGUID = packed record
D1: Longword;
D2: Word;
D3: Word;
D4: array[0..7] of Byte;
end;
```

هرگاه ثابت نوعداری از نوع *TGUID* اعلان کنید، می‌توانید از یک لیترال رشته برای تعیین مقدار آن استفاده کنید. برای مثال،

```
const IID_IMalloc: TGUID = '{00000002-0000-0000-C000-000000000046}';
```

در فراخوان‌های روال و تابع، یا یک GUID یا یک شناسه واسط می‌توانند به صورت یک مقدار یا پارامتر ثابت از نوع *TGUID* به کار روند. برای مثال، با اعلان‌های داده شده زیر

```
function Supports(Unknown: IInterface; const IID: TGUID): Boolean;
```

Supports می‌تواند به یکی از روش‌هایی که می‌آید، اعلان شود:

```
if Supports(Allocator, IMalloc) then ...
if Supports(Allocator, IID_IMalloc) then ...
```

قراردادهای فراخوانی برای واسط‌ها

قرارداد فراخوانی پیش فرض **register** است، اما واسط‌های به اشتراک گذاشته شده در میان مدول‌ها (به ویژه اگر آنها در زبان‌های متفاوتی نوشته شده باشند) بایستی همه متدها را با **stdcall** اعلان کنند. برای پیاده‌سازی واسط‌های CORBA از **safecall** استفاده کنید. در ویندوز، شما می‌توانید از **safecall** برای پیاده‌سازی متدهای واسط‌های دوگانه استفاده کنید (بخش «واسط‌های دوگانه (تنها برای ویندوز)» را در همین فصل ملاحظه نمایید). برای آگاهی از اطلاعات بیشتر درباره قراردادهای فراخوانی، بخش «قراردادهای فراخوانی» را در فصل ۶ ملاحظه نمایید.

خاصیت‌های واسط

خاصیت‌های اعلان شده در یک واسط تنها از طریق عباراتی از نوع واسط قابل دسترسی هستند؛ آنها نمی‌توانند از طریق متغیرهای نوع کلاس در دسترس باشند. علاوه بر این، خاصیت‌های واسط تنها در میان برنامه‌ها، جایی که واسط در آنجا کامپایل شده، قابل مشاهده هستند. برای مثال، در ویندوز اشیای COM دارای هیچ خاصیتی نیستند.

در یک واسط، از آن جایی که فیلدی موجود نیست، تصریح کننده‌های **read** و **write** خاصیت بایستی متد باشند.

اعلان‌های Forward

اعلان یک واسط که با واژه کلیدی **interface** و یک نقطه ویرگول خاتمه می‌یابد، بدون تعیین یک نیا، GUID یا لیست اعضاء، یک اعلان پیشرو (*forward*) است. یک اعلان *forward* باید توسط یک تعریف اعلان از همان واسط در میان همان بخش اعلان نوع تثبیت شود. به عبارت دیگر، میان یک اعلان *forward* و تعریف اعلانش، هیچ چیزی به جز اعلان‌های نوع دیگر نمی‌تواند اتفاق بیفتد. اعلان‌های پیشرو (*forward*) اجازه واسط‌های وابسته متقابل (به صورت دوسره) را می‌دهند. برای مثال،

```
type
IControl = interface;
```

```
IWindow = interface
['{00000115-0000-0000-C000-000000000044}']
function GetControl(Index: Integer): IControl;
...
end;
IControl = interface
['{00000115-0000-0000-C000-000000000049}']
function GetWindow: IWindow;
...
end;
```

واسط‌های مشتق شده به طور متقابل، مجاز نیستند. برای مثال، اشتقاق *IWindow* از *IControl* و نیز اشتقاق *IControl* از *IWindow* قانونی نیست.

پیاده‌سازی واسط‌ها

همین که یک واسط اعلان شده باشد، قبل از این که بتواند به کار برده شود، بایستی در یک کلاس پیاده‌سازی شود. واسط‌های پیاده‌سازی شده توسط یک کلاس در اعلان کلاس، بعد از نام نیای کلاس، تعیین می‌شوند. یک چنین اعلان‌هایی قالب زیر را دارند

```
type className = class (ancestorClass, interface1, ..., interfacen)
memberList
end;
```

برای مثال،

```
type
TMemoryManager = class(TInterfacedObject, IMalloc, IErrorInfo)
...
end;
```

کلاسی به نام *TMemoryManager* اعلان می‌کند که واسط‌های *IMalloc* و *IErrorInfo* را پیاده‌سازی می‌کند. هرگاه یک کلاس واسطی را پیاده‌سازی کند، باید هر متد اعلان شده در واسط را پیاده‌سازی کند (یا یک پیاده‌سازی از هر متد اعلان شده در واسط را به ارث ببرد).

در اینجا اعلان *TInterfacedObject* واقع در یونیت *System*، آورده شده است.

```
type
TInterfacedObject = class(TObject, IInterface)
protected
FRefCount: Integer;
function QueryInterface(const IID: TGUID; out Obj): HResult; stdcall;
function _AddRef: Integer; stdcall;
function _Release: Integer; stdcall;
public
procedure AfterConstruction; override;
procedure BeforeDestruction; override;
```

```
class function NewInstance: TObject; override;
property RefCount: Integer read FRefCount;
end;
```

TInterfacedObject واسط *IInterface* را پیاده‌سازی می‌کند. از این رو *TInterfacedObject* هر سه متد *IInterface* را اعلان و پیاده‌سازی می‌کند.

در ضمن کلاس‌هایی که واسط‌ها را پیاده‌سازی می‌کنند، می‌توانند به عنوان کلاس‌های پایه به کار برده شوند. (اولین مثال بالا *TMemoryManager* را به عنوان یک فرزند مستقیم *TInterfacedObject* اعلان می‌کند.) از آن جایی که هر واسط از *IInterface* ارث می‌برد، یک کلاس که واسط‌ها را پیاده‌سازی می‌کند باید متدهای *QueryInterface*، *_AddRef* و *_Release* را پیاده‌سازی کند. *TInterfacedObject* واقع در یونیت *System* این متدها را پیاده‌سازی می‌کند و از این رو یک کلاس پایه سرراست و مناسبی است تا از آن کلاس‌های دیگر که واسط‌ها را پیاده‌سازی می‌کنند مشتق شوند.

هرگاه یک واسط پیاده‌سازی شود، هر یک از متدهای آن به یک متد واقع در کلاس پیاده‌سازی نگاشته می‌شود که نوع نتیجه مشابه، قرارداد فراخوانی مشابه، تعداد پارامترهای مشابه و پارامترهای نوعدار شده به طور یکسان در هر موقعیت خواهد داشت. به طور پیش فرض، هر متد واسط به یک متد از همان نام در کلاس پیاده‌سازی نگاشته می‌شود.

شروط وضوح متد

شما می‌توانید نگاشت‌های مبتنی بر اسم را با ضمیمه کردن شروط وضوح متد در یک اعلان کلاس، باطل کنید. هرگاه یک کلاس دو یا چند واسط را که متدهایی با نام‌های یکسان دارند، پیاده‌سازی می‌کند، از شروط وضوح متد برای حل تعارضات نام‌گذاری استفاده کنید. یک شرط وضوح متد قالب زیر را دارد

```
procedure interface.interfaceMethod = implementingMethod;
```

یا

```
function interface.interfaceMethod = implementingMethod;
```

جایی که *implementingMethod* یک متد اعلان شده در کلاس یا یکی از نیاهایش است. *implementingMethod* می‌تواند یک متد باشد که بعداً در اعلان کلاس، اعلان می‌شود، اما نمی‌تواند

یک متد خصوصی (private) یک کلاس نیای اعلان شده در مدول دیگری باشد. برای مثال، اعلان کلاس

```
type
TMemoryManager = class(TInterfacedObject, IMalloc, IErrorInfo)
function IMalloc.Alloc = Allocate;
procedure IMalloc.Free = Deallocate;
...
end;
```

متدهای *Alloc* و *Free* مال *IMalloc* را به متدهای *Allocate* و *Deallocate* متعلق به *TMemoryManager* می‌نگارد.

یک شرط وضوح متد نمی‌تواند یک نگاشت معرفی شده توسط یک کلاس نیا را تغییر دهد.

تغییر پیاده‌سازی‌های موروثی

کلاس‌های فرزند می‌توانند روشی را که یک متد به خصوص پیاده‌سازی می‌شود، با ابطال و تعریف مجدد پیاده‌سازی متد تغییر دهند. لازمه این کار این است که پیاده‌سازی متد مجازی یا دینامیک باشد. در ضمن یک کلاس می‌تواند یک واسط یکپارچه را که از یک کلاس نیا ارث می‌برد، پیاده‌سازی مجدد کند. این کار مستلزم لیست‌بندی مجدد واسط در اعلان کلاس فرزند است. برای مثال،

```
type
IWindow = interface
['{00000115-0000-0000-C000-000000000146}']
procedure Draw;
...
end;
TWindow = class(TInterfacedObject, IWindow)     // TWindow implements IWindow
procedure Draw;
...
end;
TFrameWindow = class(TWindow, IWindow)          // TFrameWindow reimplements IWindow
procedure Draw;
...
end;
```

پیاده‌سازی مجدد یک واسط پیاده‌سازی موروثی همان واسط را پنهان می‌کند. از این رو شروط وضوح متد در یک کلاس نیا تأثیری بر واسطی که مجدداً پیاده‌سازی شده، ندارند.

پیاده‌سازی واسط‌ها به نیابت

راهنمای **implements** به شما اجازه می‌دهد تا پیاده‌سازی یک واسط را به یک خاصیت واقع در پیاده‌سازی کلاس، محول کنید. برای مثال،

```
property MyInterface: IMyInterface read FMyInterface implements IMyInterface;
```

یک خاصیت به نام *MyInterface* اعلان می‌کند که واسط *IMyInterface* را پیاده‌سازی می‌کند.

راهنمای **implements** بایستی آخرین تصریح کننده در اعلان خاصیت باشد و می‌توان بیشتر از یک واسط را که با ویرگول از هم جدا شده‌اند، لیست کرد. خاصیت نایب

- باید از یک نوع کلاس یا واسط باشد.
- نمی‌تواند یک خاصیت آرایه‌ای باشد یا یک تصریح کننده **index** داشته باشد.
- باید یک تصریح کننده **read** داشته باشد. اگر خاصیت از یک متد **read** استفاده می‌کند، آن متد باید از قرارداد فراخوانی پیش فرض **register** استفاده کند و نمی‌تواند دینامیک باشد (گرچه می‌تواند مجازی باشد) یا راهنمای **message** را تصریح کند.

توجه کلاسی که شما برای پیاده‌سازی واسط محول شده استفاده می‌کنید بایستی از *TAggregatedObject* مشتق شود.

نیابت دادن به یک خاصیت نوع واسط

چنان چه خاصیت نایب از یک نوع واسط باشد، آن واسط، یا واسطی که از آن مشتق شده است، باید در فهرست نیای کلاس جایی که خاصیت اعلان شده است، ظاهر شود. خاصیت نایب باید یک شیء برگرداند که کلاسش به طور کامل واسط مشخص شده توسط راهنمای **implements** را پیاده‌سازی کند و ضمناً این کار را بدون شروط وضوح متد انجام دهد. برای مثال،

```
type
IMyInterface = interface
procedure P1;
procedure P2;
end;
TMyClass = class(TObject, IMyInterface)
FMyInterface: IMyInterface;
property MyInterface: IMyInterface read FMyInterface implements IMyInterface;
end;
var
MyClass: TMyClass;
MyInterface: IMyInterface;
begin
MyClass := TMyClass.Create;
MyClass.FMyInterface := ...        // some object whose class implements IMyInterface
MyInterface := MyClass;
MyInterface.P1;
end;
```

نیابت دادن به یک خاصیت نوع کلاس

چنان چه خاصیت نایب از یک نوع کلاس باشد، قبل از این که کلاس محیطی و نیاکانش وارسی شوند، آن کلاس و نیاکانش برای پیاده‌سازی متدهای واسط مشخص شده جستجو می‌شوند. از این رو پیاده‌سازی برخی متدها در کلاس مشخص شده بوسیله خاصیت، و دیگر متدهای واقع در کلاس جایی که خاصیت اعلان شده، امکان‌پذیر است.

شروط وضوح متد می‌توانند در روش معمول برای حل ابهامات یا تعیین یک متد به خصوص به کار برده شوند. یک واسط نمی‌تواند توسط بیشتر از یک خاصیت نوع کلاس پیاده‌سازی شود. برای مثال،

```
type
IMyInterface = interface
procedure P1;
procedure P2;
end;
TMyImplClass = class
procedure P1;
procedure P2;
end;
TMyClass = class(TInterfacedObject, IMyInterface)
FMyImplClass: TMyImplClass;
property MyImplClass: TMyImplClass read FMyImplClass implements IMyInterface;
procedure IMyInterface.P1 = MyP1;
procedure MyP1;
end;
procedure TMyImplClass.P1;
...
procedure TMyImplClass.P2;
...
procedure TMyClass.MyP1;
...
var
MyClass: TMyClass;
MyInterface: IMyInterface;
begin
MyClass := TMyClass.Create;
MyClass.FMyImplClass := TMyImplClass.Create;
MyInterface := MyClass;
MyInterface.P1; // calls TMyClass.MyP1;
MyInterface.P2; // calls TImplClass.P2;
end;
```

ارجاعات واسط

چنان چه متغیری از یک نوع واسط را اعلان کنید، متغیر می‌تواند به وهله‌های هر کلاسی که واسط را پیاده‌سازی می‌کند، ارجاع کند. یک چنین متغیرهایی به شما اجازه می‌دهند تا متدهای واسط را بدون آگاه شدن از این که در زمان کامپایل واسط در کجا پیاده‌سازی شده است، فراخوانی کنید. اما آنها مشروط به محدودیت‌های زیر هستند.

- یک عبارت نوع واسط تنها امکان دسترسی به متدها و خاصیت‌های اعلان شده در واسط، و نه اعضای دیگر کلاس پیاده‌سازی را به شما می‌دهد.
- یک عبارت نوع واسط نمی‌تواند به یک شیء که کلاسش یک واسط فرزند را پیاده‌سازی می‌کند، ارجاع کند، مگر این که کلاس (یا یکی که کلاس از آن ارث می‌برد) به طور ضمنی واسط نیا را هم پیاده‌سازی کند.

برای مثال،

```
type
IAncestor = interface
end;
IDescendant = interface(IAncestor)
procedure P1;
end;
TSomething = class(TInterfacedObject, IDescendant)
procedure P1;
procedure P2;
end;
...
var
D: IDescendant;
A: IAncestor;
begin
D := TSomething.Create; // works!
A := TSomething.Create; // error
D.P1; // works!
D.P2; // error
end;
```

در این مثال،

- A به عنوان متغیری از نوع *IAncestor* اعلان شده است. از آن جایی که *TSomething*، *IAncestor* را همراه با واسط‌هایی که پیاده‌سازی می‌کند لیست نمی‌کند، وهله‌ای از *TSomething* نمی‌تواند به A تخصیص داده شود. اما چنان چه اعلان *TSomething* را به شکل زیر تغییر می‌دادیم

```
TSomething = class(TInterfacedObject, IAncestor, IDescendant)
...
```

اولین خطا یک تخصیص معتبر می‌شد.

- D به عنوان متغیری از نوع *IDescendant* اعلان شده است. در حالی که D به وهله‌ای از *TSomething* ارجاع می‌کند، ما نمی‌توانیم آن را برای دسترسی به متد *P2* از *TSomething* به کار ببریم، زیرا *P2* یک متد *IDescendant* نیست. اما چنان چه اعلان D را به شکل زیر تغییر می‌دادیم

```
D: TSomething;
```

دومین خطا یک فراخوان متد معتبر می‌شد.

ارجاعات واسط از طریق شمارش–ارجاعات مدیریت می‌شوند، که به متدهای *_AddRef* و *_Release* ارث رسیده از *IInterface* بستگی دارد. هرگاه به یک شیء تنها از طریق واسط‌ها ارجاع شود، دیگر نیاز نیست که آن را به طور دستی تخریب کرد؛ زمانی که آخرین ارجاع به آن از دامنه بیرون افتاد، شیء به طور خودکار تخریب می‌شود. متغیرهای سراسری نوع واسط تنها می‌توانند با **nil** مقداردهی اولیه شوند. برای این که مشخص شود که آیا یک عبارت نوع واسط به یک شیء ارجاع می‌کند، آن را به تابع استاندارد *Assigned* ارسال کنید.

سازگاری برای تخصیص واسط

یک نوع کلاس سازگار برای تخصیص با هر نوع واسط پیاده‌سازی شده توسط کلاس است. یک نوع واسط سازگار برای تخصیص با هر نوع واسط نیاست. مقدار **nil** می‌تواند به هر متغیر نوع واسط تخصیص داده شود.

عبارتی از نوع واسط می‌تواند به یک واریانت تخصیص داده شود. اگر واسط از نوع *IDispatch* یا یک فرزند باشد، واریانت کد نوع *varDispatch* را اخذ می‌کند. در غیر این صورت، واریانت کد نوع *varUnknown* را دریافت می‌کند.

یک واریانت که کد نوعش *varEmpty*، *varUnknown* یا *varDispatch* باشد، می‌تواند به یک متغیر *IInterface* تخصیص داده شود. یک واریانت که کد نوعش *varEmpty* یا *varDispatch* باشد می‌تواند به یک متغیر *IDispatch* تخصیص داده شود.

قالب‌بندی‌های واسط

نوع‌های واسط از همان قواعد نوع‌های کلاس در قالب‌بندی‌های نوعی و مقداری پیروی می‌کنند. عبارت‌های نوع کلاس می‌توانند به نوع واسط قالب‌بندی (تبدیل نوع صریح) شوند— برای مثال، IMyInterface(SomeObject)— به شرط آن که کلاس واسط را پیاده‌سازی کند.

یک عبارت نوع واسط می‌تواند به *Variant* قالب‌بندی شود. اگر واسط از نوع *IDispatch* یا یک فرزند باشد، نتیجه واریانت کد نوع *varDispatch* را خواهد داشت. در غیر این صورت، نتیجه واریانت کد نوع *varUnknown* را خواهد داشت.

یک واریانت که کد نوعش *varEmpty*، *varUnknown* یا *varDispatch* باشد، می‌تواند به *IInterface* قالب‌بندی شود. یک واریانت که کد نوعش *varEmpty* یا *varDispatch* باشد می‌تواند به *IDispatch* قالب‌بندی شود.

جستار واسط

شما می‌توانید عملگر **as** را برای اجرای قالب‌بندی‌های نوع بررسی شده به کار ببرید. این کار به عنوان جستار واسط[۱] شناخته می‌شود و عبارت نوع واسطی از یک ارجاع شیء یا از ارجاع واسط دیگری مبتنی بر نوع (زمان اجرا) واقعی شیء را، تسلیم می‌کند. یک جستار واسط قالب زیر را دارد

```
object as interface
```

جایی که *object* عبارتی از یک نوع واسط یا واریانت بوده یا وهله‌ای از یک کلاس را که یک واسط را پیاده‌سازی می‌کند، مشخص می‌کند و *interface* هر واسط اعلان شده با یک GUID است.

اگر *object* برابر **nil** باشد یک جستار واسط **nil** را برمی‌گرداند. در غیر این صورت، GUID مربوط به *interface* را به متد *QueryInterface* در *object* ارسال می‌کند، همراه با تولید یک اخطار مگر اینکه *QueryInterface* صفر را برگرداند. اگر *QueryInterface* صفر را برگرداند (نشانگر این که کلاس *object*، *interface* را پیاده‌سازی می‌کند)، جستار واسط یک ارجاع واسط به *object* را برمی‌گرداند.

اشیاء اتوماسیون (تنها برای ویندوز)

[۱] Interface querying

شیئی که کلاسش واسط *IDispatch* (اعلان شده در یونیت *System*) را پیاده‌سازی می‌کند، یک شیء اتوماسیون[۱] است. اتوماسیون تنها بر روی ویندوز در دسترس است.

نوع‌های واسط توزیع (تنها برای ویندوز)

نوع‌های واسط توزیع[۲] متدها و خاصیت‌هایی را تعریف می‌کنند که یک شیء اتوماسیون را از طریق *IDispatch* پیاده‌سازی می‌کنند. در زمان اجرا فراخوان‌ها به متدهای یک واسط توزیع از طریق متد *Invoke* متعلق به *IDispatch*، تعیین مسیر می‌شوند؛ یک کلاس نمی‌تواند یک واسط توزیع را پیاده‌سازی کند. اعلان یک نوع واسط توزیع قالب زیر را دارد

```
type interfaceName = dispinterface
['{GUID}']
memberList
end;
```

جایی که ['{*GUID*}'] اختیاری بوده و *memberList* از اعلان‌های خاصیت و متد تشکیل می‌شود. اعلان‌های واسط توزیع مشابه اعلان معمولی واسط‌ها هستند، اما نمی‌توانند نیایی را مشخص کنند. برای مثال،

```
type
IStringsDisp = dispinterface
['{EE05DFE2-5549-11D0-9EA9-0020AF3D82DA}']
property ControlDefault[Index: Integer]: OleVariant dispid 0; default;
function Count: Integer; dispid 1;
property Item[Index: Integer]: OleVariant dispid 2;
procedure Remove(Index: Integer); dispid 3;
procedure Clear; dispid 4;
function Add(Item: OleVariant): Integer; dispid 5;
function _NewEnum: IUnknown; dispid -4;
end;
```

متدهای واسط توزیع (تنها برای ویندوز)

متدهای یک واسط توزیع نمونه‌های اولیه‌ای برای فراخوان‌ها به متد *Invoke* پیاده‌سازی متضمن *IDispatch* هستند. برای تعیین یک ID توزیع اتوماسیون برای یک متد، راهنمای **dispid** را در اعلان

---

[۱] Automation object

[۲] Dispatch interface types

متد جای دهید، که این راهنما با یک ثابت عددی صحیح پی گرفته می‌شود؛ تعیین یک ID که قبلاً به کار برده شده، سبب بروز یک خطا می‌شود.

یک متد اعلان شده در یک واسط توزیع نمی‌تواند حاوی راهنماهایی به غیر از **dispid** باشد. نوع‌های پارامتر و نتیجه باید قابل automate شدن باشند— یعنی، آنها باید *Byte*، *Currency*، *Real*، *Double*، *Longint*، *Integer*، *Single*، *Smallint*، *AnsiString*، *WideString*، *TDateTime*، *Variant*، *OleVariant*، *WordBool* یا هر گونه نوع واسط باشند.

خاصیت‌های واسط توزیع

خاصیت‌های یک واسط توزیع حاوی تصریح کننده‌های دسترسی نیستند. آنها می‌توانند به صورت **readonly** یا **writeonly** اعلان شوند. برای تعیین یک ID توزیع برای یک خاصیت، راهنمای **dispid** را در اعلانش جای دهید، که این راهنما با یک ثابت عددی صحیح پی گرفته می‌شود؛ تعیین یک ID که قبلاً مورد استفاده قرار گرفته است، سبب بروز یک خطا می‌شود. خاصیت‌های آرایه می‌توانند به صورت **default** اعلان شوند. هیچ راهنمای دیگری در اعلان‌های خاصیت واسط توزیع، مجاز نیستند.

دسترسی به اشیای اتوماسیون (تنها برای ویندوز)

از واریانت‌ها برای دسترسی به اشیای اتوماسیون استفاده کنید. هرگاه یک واریانت به یک شیء اتوماسیون ارجاع کند، شما می‌توانید متدهای شیء را فراخوانی کنید و خاصیت‌های آن را از طریق واریانت خوانده و تغییر دهید. برای انجام این کار، شما باید *ComObj* را در شرط **uses** یکی از یونیت‌های خود یا برنامه یا کتابخانه‌تان قرار دهید.

فراخوان‌های متد شیء اتوماسیون در زمان اجرا مقید شده هستند و به هیچ اعلان متد قبلی نیاز ندارند. صحت و اعتبار این فراخوان‌ها در زمان کامپایل بررسی نمی‌شوند.

مثال زیر فراخوان‌های متد اتوماسیون را توضیح می‌دهد. تابع *CreateOleObject* (تعریف شده در *ComObj*) یک ارجاع *IDispatch* به یک شیء اتوماسیون برمی‌گرداند و سازگار برای تخصیص با متغیر *Word* است.

```
var
Word: Variant;
begin
Word := CreateOleObject('Word.Basic');
```

```
Word.FileNew('Normal');
Word.Insert('This is the first line'#13);
Word.Insert('This is the second line'#13);
Word.FileSaveAs('c:\temp\test.txt', 3);
end;
```

شما می‌توانید پارامترهای نوع واسط را به متدهای اتوماسیون ارسال کنید.

آرایه‌های واریانت با یک نوع عنصر *varByte* متد ترجیحی برای ارسال داده‌های باینری در میان کنترل کننده‌ها و سرورهای اتوماسیون هستند. یک چنین آرایه‌هایی در معرض این هستند که هیچ انتقال داده‌ای از آنها صورت نگیرد و می‌توانند با استفاده از روتین‌های *VarArrayLock* و *VarArrayUnlock* به طور مؤثری در دسترس باشند.

ترکیب نوشتاری فراخوان متد شیء اتوماسیون

ترکیب نوشتاری/نحوی فراخوان یک متد شیء اتوماسیون یا دسترسی خاصیت اتوماسیون با یک فراخوانی متد نرمال یا دسترسی خاصیت نرمال مشابه است. گرچه، فراخوان‌های متد اتوماسیون می‌توانند از هر دو پارامتر موقعیتی[۱] یا اسمی[۲] استفاده کنید. (اما برخی سرورهای اتوماسیون از پارامترهای اسمی پشتیبانی نمی‌کنند.)

یک پارامتر موقعیتی در واقع یک عبارت است. یک پارامتر اسمی از یک شناسه پارامتر که با علامت **:=** پی گرفته شده و بعد از آن هم یک عبارت می‌آید، تشکیل می‌شود. پارامترهای موقعیتی باید مقدم بر هر پارامتر اسمی در فراخوان یک متد باشند. پارامترهای اسمی می‌توانند به هر ترتیبی تعیین گردند.

برخی سرورهای اتوماسیون به شما اجازه می‌دهند تا پارامترهایی را از فراخوان یک متد، با پذیرش مقادیر پیش فرض آنها، دور بیاندازید. برای مثال،

```
Word.FileSaveAs('test.doc');
Word.FileSaveAs('test.doc', 6);
Word.FileSaveAs('test.doc',,,'secret');
Word.FileSaveAs('test.doc', Password := 'secret');
Word.FileSaveAs(Password := 'secret', Name := 'test.doc');
```

پارامترهای فراخوان متد اتوماسیون می‌توانند از انواع صحیح، حقیقی، رشته، بولی و واریانت باشند. چنان چه عبارت پارامتر تنها از یک ارجاع متغیر تشکیل شده باشد، و اگر ارجاع متغیر از نوع *Byte*،

---

[۱] positional

[۲] named

Smallint، *Integer*، *Single*، *Double*، *Currency*، *TDateTime*، *AnsiString*، *WordBool* یـا *Variant* باشـد، یک پارامتر به واسطه ارجاع (by reference) ارسال می‌شود. اگر عبارت از یکی از این نوع‌ها نباشد، یا اگر صرفاً یک متغیر نباشد، پارامتر به واسطه مقدار (by value) ارسال می‌شود. ارسال یـک پـارامتر بـه واسطه ارجاع (by reference) به یک متد که منتظر یک پارامتر مقدار اسـت سـبب مـی‌شـود کـه COM مقدار را از پارامتر ارجاعی اخذ کند. ارسال یک پارامتر به واسطه مقدار (by value) به متدی که منتظـر یک پارامتر ارجاعی است سبب بروز یک خطا می‌شود.

واسط‌های دوگانه (تنها برای ویندوز)

یک واسط دوگانه، واسطی است که هم انقیـاد[۱] زمـان کامپایـل و هـم انقیـاد زمـان اجـرا را از طریـق اتوماسیون پشتیبانی می‌کند. واسط‌های دوگانه باید از *IDispatch* مشتق شوند.

تمامی‌متدهای یک واسط دوگانه (به جز آنهایی که از *IInterface* و *IDispatch* به ارث می‌رسند) باید از قرادادِ **safecall** استفاده کنند و همه انواع پارامتر و نتیجه متد باید قابـل خودکارسـازی باشـند. (انـواع قابل automate شدن *Byte*، *Currency*، *Real*، *Double*، *Real48*، *Integer* ، *Single* ، *Smallint* ، *AnsiString* ، *ShortString*، *TDateTime*، *Variant*، *OleVariant* و *WordBool* هستند.)

---

[۱] binding

فصل ۱۱

مدیریت حافظه

این فصل چگونگی استفاده برنامه‌ها از حافظه را توضیح می‌دهد و فرمت‌های درونی انواع داده پاسکال شیئی را شرح می‌دهد.

مدیر حافظه (تنها برای ویندوز)

توجه لینوکس از توابع glibc از قبیل *malloc* برای مدیریت حافظه استفاده می‌کند. برای اطلاعات بیشتر، به صفحه اصلی *malloc* در سیستم لینوکس خود مراجعه کنید.

در سیستم‌های ویندوز، در یک برنامه مدیر حافظه تمامی تخصیصات و آزادسازی‌های حافظه پویا را مدیریت می‌کند. روال‌های استاندارد *New*، *Dispose*، *GetMem*، *ReallocMem* و *FreeMem* از مدیر حافظه استفاده می‌کنند و تمامی اشیاء و رشته‌های بلند از طریق مدیر حافظه تخصیص حافظه می‌شوند.

در ویندوز، مدیر حافظه برای برنامه‌هایی که تعداد زیادی از بلوک‌های اندازه کوچک تا متوسط حافظه را تخصیص می‌دهند، بهینه سازی شده است، همان طور که برای برنامه‌های شیءگرا و برنامه‌هایی که داده‌های رشته‌ای را پردازش می‌کنند معمول و رایج است. مدیرهای حافظه دیگر، مانند پیاده‌سازی‌های

GlobalAlloc، *LocalAlloc* و پشتیبان هیپ خصوصی (private heap) در ویندوز، نوعاً در چنین موقعیت‌هایی به خوبی کار نمی‌کنند، و چنان چه به طور مستقیم استفاده می‌شدند، برنامه را کند می‌کردند.

به منظور اطمینان از بهترین عملکرد، مدیر حافظه به طور مستقیم با API حافظه مجازی Win32 میانجی‌گری می‌کند (توابع *VirtualAlloc* و *VirtualFree*). مدیر حافظه، حافظه را از سیستم عامل در بخش‌های 1-MB فضای آدرس رزرو می‌کند و حافظه را هر اندازه که مورد نیاز باشد در نمونه‌های شانزده کیلوبایتی (16-KB) به کار می‌گیرد. مدیر حافظه، حافظه استفاده نشده در بخش‌های 16-KB و 1-MB را رها کرده و ترخیص می‌کند. برای بلوک‌های کوچکتر، حافظه به کار گرفته شده مجدداً زیر تخصیص داده شده می‌شود.

بلوک‌های مدیر حافظه همواره تا مرز ۴-بایت گرد می‌شوند و همیشه شامل یک هدر ۴-بایت هستند که در آن اندازه بلوک و بیت‌های وضعیت دیگر ذخیره می‌شوند. این سخن به معنای این است که بلوک‌های مدیر حافظه همواره به صورت دو کلمه‌ای ردیف می‌شوند (double-word-aligned)، که هنگام آدرس‌دهی بلوک، عملکرد بهینه CPU را تضمین می‌کند.

مدیر حافظه دو متغیر وضعیت را نگه می‌دارد: *AllocMemCount*، *AllocMemSize*، که حاوی تعداد بلوک‌های حافظه تخصیص یافته جاری و اندازه ترکیب شده تمامی بلوک‌های حافظه تخصیص یافته به طور جاری هستند. برنامه می‌تواند از این متغیرها برای نمایش اطلاعات وضعیت مربوط به اشکال‌زدایی[۱] استفاده کند.

یونیت *System* دو روال ارائه می‌کند: *GetMemoryManager* و *SetMemoryManager*، که به برنامه‌ها اجازه می‌دهند تا از فراخوان‌های تراز پایین مدیر حافظه جلوگیری کنند. در ضمن یونیت *System* تابعی به نام *GetHeapStatus* عرضه می‌کند که یک رکورد محتوی اطلاعات جزییاتی وضعیت مدیر حافظه را برمی‌گرداند.

---

[۱] debugging

متغیرها

متغیرهای سراسری در سگمنت داده برنامه تخصیص حافظه می‌شوند و تا مدتی که برنامه طول می‌کشد، دوام دارند. متغیرهای محلی (اعلان شده در میان روال‌ها و توابع) در پشته[۱] یک برنامه جای می‌گیرند. هر زمان که یک روال یا تابع فراخوان می‌شود، مجموعه‌ای از متغیرهای محلی را تخصیص حافظه می‌دهد؛ هنگام خروج، متغیرهای محلی آزاد می‌شوند. بهینه‌سازی کامپایلر ممکن است متغیرها را خیلی زودتر حذف کرده باشد.

توجه در لینوکس، اندازه پشته صرفاً توسط محیط تنظیم می‌شود.

در ویندوز، پشته یک برنامه توسط دو مقدار تعریف می‌شود: اندازه پشته حداقل و اندازه پشته حداکثر. این مقادیر از طریق راهنماهای کامپایلر **$MINSTACKSIZE** و **$MAXSTACKSIZE** کنترل می‌شوند و مقدار پیش فرضشان به ترتیب برابر 16,384 (16K) و 1,048,576 (1M) هستند. تضمین شده است که یک برنامه، اندازه پشته حداقلی داشته باشد، و پشته یک برنامه هرگز مجاز نیست که از اندازه پشته حداکثر بالاتر رود. اگر حافظه کافی برای ارضای پشته حداقل مورد نیاز یک برنامه موجود نباشد، ویندوز به مجرد تلاشی برای آغاز برنامه یک خطا گزارش می‌کند.

اگر یک برنامه ویندوز نیازمند فضای پشته بیشتری از اندازه پشته حداقل مشخص شده باشد، حافظه اضافی به طور خودکار در نموهای 4K تخصیص داده می‌شود. اگر تخصیص فضای پشته اضافی نافرجام بماند، به علت این که یا حافظه بیشتری در دسترس نباشد یا اندازه کل پشته از اندازه پشته حداکثر تجاوز کند، یک استثنای *EStackOverflow* تولید می‌شود. (بررسی سرریز پشته به طور کامل خودکار است. راهنمای کامپایلر **$S**، که در اصل بررسی سرریز کنترل شده است، برای سازگاری با گذشته پشتیبانی می‌شود.)

در ویندوز یا لینوکس، متغیرهای پویای ایجاد شده با روال‌های *GetMem* یا *New* در پشته تخصیص حافظه می‌شوند و تا زمانی که با *FreeMem* یا *Dispose* بازستانی نشوند، استمرار خواهند داشت.

---

[۱] Stack

رشته‌های بلند، رشته‌های پهن، آرایه‌های پویا، واریانت‌ها و واسط‌ها در پشته تخصیص حافظه می‌شوند، اما حافظه آنها به طور خودکار مدیریت می‌شود.

فرمت درونی داده‌ها

بخش‌های بعدی فرمت‌های درونی انواع داده پاسکال شیئی را شرح می‌دهند.

قبل از فراگیری مطالب این بخش به یاد داشته باشید که هر بایت برابر ۸ بیت است و هر کلمه برابر ۲ بایت یا ۱۶ بیت است.

1 Byte = 8 Bit
1 Word = 2 Byte = 16 Bit

انواع صحیح

فرمت متغیری از نوع صحیح بستگی به حدود بالا و پایین آن دارد.

- اگر هردو حد بالا و پایین در دامنه –128..127 (Shortint) باشند، متغیر به صورت یک بایت علامت‌دار ذخیره می‌شود.
- اگر هردو حد بالا و پایین در دامنه 0..255 (Byte) باشند، متغیر به صورت یک بایت بدون علامت ذخیره می‌شود.
- اگر هردو حد بالا و پایین در دامنه –32768..32767 (Smallint) باشند، متغیر به صورت یک کلمه علامت‌دار ذخیره می‌شود.
- اگر هردو حد بالا و پایین در دامنه 0..65535 (Word) باشند، متغیر به صورت یک کلمه بدون علامت ذخیره می‌شود.
- اگر هردو حد بالا و پایین در دامنه –2147483648..2147483647 (Longint) باشند، متغیر به صورت یک کلمه مضاعف علامت‌دار ذخیره می‌شود.
- اگر هردو حد بالا و پایین در دامنه 0..4294967295 (Longword) باشند، متغیر به صورت یک کلمه مضاعف بدون علامت ذخیره می‌شود.
- در غیر این صورت، متغیر همانند یک کلمه چهارگانه علامت‌دار ذخیره می‌شود (Int64).

انواع کاراکتر

یک *Char*، یک *AnsiChar* یا زیردامنه ای از یک نوع *Char* همانند یک بایت بدون علامت ذخیره می‌شوند. یک *WideChar* همانند یک کلمه بدون علامت ذخیره می‌شود.

انواع بولی (Boolean)

یک نوع *Boolean* به صورت یک *Byte* ذخیره می‌شود، یک *ByteBool* به صورت یک *Byte* ذخیره می‌شود، یک نوع *WordBool* به صورت یک *Word* ذخیره می‌شود و یک *LongBool* به صورت یک *Longint* ذخیره می‌شود.

یک *Boolean* می‌تواند مقادیر صفر (*False*) و یک (*True*) را به خود بگیرد. انواع *ByteBool*، *WordBool* و *LongBool* می‌توانند مقادیر صفر (*False*) و غیرصفر (*True*) را به خود بگیرند.

انواع شمارشی

چنان چه برشمارشی بیشتر از ۲۵۶ مقدار نداشته باشد و نوع در حالت **{$Z1}** (پیش فرض) اعلان شده باشد، یک نوع شمارشی به صورت یک بایت بدون علامت ذخیره می‌شود. اگر یک نوع شمارشی بیشتر از ۲۵۶ مقدار داشته باشد، یا اگر نوع در حالت **{$Z2}** اعلان شده باشد، به صورت یک کلمه بدون علامت ذخیره می‌شود. چنان چه یک نوع شمارشی در حالت **{$Z4}** اعلان شده باشد، به صورت یک کلمه مضاعف بدون علامت ذخیره می‌شود.

انواع حقیقی

انواع حقیقی نمایش باینری یک علامت (+ یا −)، یک نما (*exponent*) و یک *significand* را ذخیره می‌کنند. یک مقدار حقیقی قالب زیر را دارد

$$+/- significand * 2^{exponent}$$

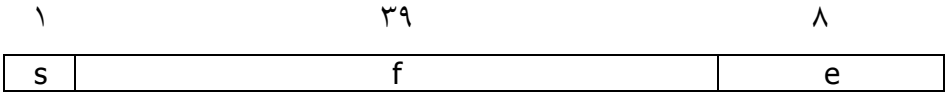
جایی که *significand* بیت واحدی نسبت به سمت چپ نقطه اعشاری باینری دارد.

(یعنی، 0 <= *significand* < 2)

در شکل‌هایی که می‌آید، بامعنی‌ترین بیت‌ها همواره در چپ قرار دارند و کم ارزش‌ترین بیت‌ها در راست. اعداد بالایی پهنای (به بیت) هر فیلد را با چپ‌ترین گزینه‌های ذخیره شده در بالاترین آدرس‌ها، نشان می‌دهند. برای مثال، برای یک مقدار *Real48*، e در اولین بایت ذخیره می‌شود، f در پنج بایت بعدی و s در بامعنی‌ترین بیت بایت آخر ذخیره می‌شود.

نوع Real48

یک عدد *Real48* ۶-بایت (۴۸-بیت) به سه فیلد تقسیم می‌شود:



اگر $0 < e <= 255$ باشد، مقدار *v* برای عدد توسط عبارت زیر داده می‌شود

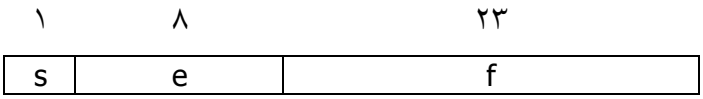
$$v = (-1)^s * 2^{(e-129)} * (1.f)$$

چنان چه $e = 0$ باشد، در این صورت $v = 0$ است.

نوع *Real48* نمی‌تواند اعداد غیرطبیعی، NaN[1]ها و بی نهایت‌ها را ذخیره کند. غیرطبیعی‌ها به هنگام ذخیره شدن در یک *Real48*، صفر می‌شوند، در حالی که اگر سعی شود NaNها و بی نهایت‌ها را در یک *Real48* ذخیره کرد، یک خطای سرریز تولید می‌شود.

نوع Single

یک عدد *Single* ۴-بایت (۳۲-بیت) به سه فیلد تقسیم می‌شود:



مقدار *v* عدد به صورت زیر داده می‌شود

اگر $0 < e < 255$ باشد، در این صورت $v = (-1)^s * 2^{(e-127)} * (1.f)$ است.

اگر $e = 0$ و $f <> 0$ باشد، در این صورت $v = (-1)^s * 2^{(-126)} * (0.f)$ است.

[1] Not a Number (NaN)

اگر $e = 0$ و $f = 0$ باشد، در این صورت $v = (-1)^s * 0$ است.

اگر $e = 255$ و $f = 0$ باشد، در این صورت $v = (-1)^s * Inf$ است.

اگر $e = 255$ و $f <> 0$ باشد، در این صورت v یک NaN است.

نوع Double

یک عدد *Double* ۸-بایت (۶۴-بیت) به سه فیلد تقسیم می‌شود:

۱	۱۱	۵۲
s	e	f

مقدار v عدد به صورت زیر داده می‌شود

اگر $0 < e < 2047$ باشد، در این صورت $v = (-1)^s * 2^{(e-1023)} * (1.f)$ است.

اگر $e = 0$ و $f <> 0$ باشد، در این صورت $v = (-1)^s * 2^{(-1022)} * (0.f)$ است.

اگر $e = 0$ و $f = 0$ باشد، در این صورت $v = (-1)^s * 0$ است.

اگر $e = 2047$ و $f = 0$ باشد، در این صورت $v = (-1)^s * Inf$ است.

اگر $e = 2047$ و $f <> 0$ باشد، در این صورت v یک NaN است.

نوع Extended

یک عدد *Extended* ۱۰-بایت (۸۰-بیت) به چهار فیلد تقسیم می‌شود:

۱	۱۵	۱	۶۳
s	e	i	f

مقدار v به صورت زیر داده می‌شود

اگر $0 <= e < 32767$ باشد، در این صورت $v = (-1)^s * 2^{(e-16383)} * (i.f)$ است.

اگر $e = 32767$ و $f = 0$ باشد، در این صورت $v = (-1)^s * Inf$ است.

اگر $e = 32767$ و $f <> 0$ باشد، در این صورت v یک NaN است.

نوع Comp

یک عدد *Comp* ۸-بایت (۶۴-بیت) همانند یک عدد صحیح ۶۴-بیت علامت‌دار ذخیره می‌شود.

نوع Currency

یک عدد *Currency* ۸-بایت (۶۴-بیت) همانند یک عدد صحیح مدرج و علامت‌دار با چهار رقم کم اهمیت که به طور ضمنی چهار مکان اعشاری را نمایش می‌دهد، ذخیره می‌شود.

انواع اشاره‌گر

یک نوع *Pointer* در۴ بایت همانند یک آدرس ۳۲-بیت ذخیره می‌شود. مقدار اشاره‌گر **nil** به صورت صفر ذخیره می‌شود.

انواع رشته کوتاه

یک رشته بایت‌هایی را به اندازه طول حداکثرش به اضافه یک بایت، اشغال می‌کند. اولین بایت حاوی طول دینامیک فعلی رشته است و بایت‌های بعدی حاوی کاراکترهای رشته هستند.

بایت طول و کاراکترها به صورت مقادیر بدون علامت در نظر گرفته می‌شود. طول رشته ماکزیمم ۲۵۵ کاراکتر به اضافه یک بایت طول است (**string**[255]) .

انواع رشته بلند

یک متغیر رشته بلند چهار بایت حافظه را اشغال می‌کند که حاوی یک اشاره‌گر به یک رشته تخصیص‌یافته به طور پویاست. هرگاه یک متغیر رشته بلند تهی باشد (حاوی رشته‌ای با طول صفر باشد)، اشاره‌گر رشته برابر **nil** است و هیچ حافظه پویایی با متغیر رشته مرتبط نیست. برای یک مقدار رشته غیر تهی، اشاره‌گر رشته به یک بلوک تخصیص یافته به طور پویای حافظه اشاره می‌کند که حاوی مقدار رشته به اضافه یک نشانگر طول ۳۲-بیت و یک شمار ارجاع ۳۲-بیت است.

جدول زیر آرایش بلوک حافظه یک رشته بلند را نمایش می‌دهد.

آرایش حافظه پویای رشته بلند **Table 11.1**

آفست	محتویات
–8	شمار ارجاع ۳۲–بیت
–4	طول به بایت
0..*Length* – 1	رشته کاراکتر
Length	کاراکتر NULL

کاراکتر NULL واقع در انتهای یک بلوک حافظه رشته بلند به طور خودکار توسط کامپایلر و روتین‌های اداره کننده رشته توکار نگه داری و پشتیبانی می‌شود. این امر قالب‌بندی (تبدیل نوع صریح) یک رشته بلند را به طور مستقیم به یک رشته منتهی به تهی امکان‌پذیر می‌سازد.

برای ثابت‌های رشته‌ای و لیترال‌ها، کامپایلر بلوک حافظه‌ای با آرایش یکسان را به عنوان یک رشته تخصیص یافته به طور پویا، تولید می‌کند، اما با یک شمار ارجاع 1-. هرگاه یک ثابت رشته‌ای به یک متغیر رشته بلند تخصیص داده شود، به اشاره‌گر رشته، آدرس بلوک حافظه تولید شده برای ثابت رشته‌ای تخصیص داده می‌شود. روتین‌های اداره کننده رشته توکار می‌دانند که نباید تلاشی برای تغییر بلوک‌هایی که یک شمار ارجاع 1- دارند صورت دهند.

انواع رشته پهن

توجه در لینوکس، رشته‌های پهن دقیقاً همانند رشته‌های بلند پیاده‌سازی می‌شوند.

در ویندوز، یک متغیر رشته پهن چهار بایت از حافظه را اشغال می‌کند که حاوی یک اشاره‌گر به یک رشته تخصیص یافته به طور پویاست. هرگاه یک متغیر رشته پهن تهی باشد (حاوی یک رشته با طول صفر باشد)، اشاره‌گر رشته برابر **nil** است و هیچ حافظه پویایی با متغیر رشته مرتبط نیست. برای یک مقدار رشته غیر تهی، اشاره‌گر رشته به یک بلوک تخصیص یافته به طور پویای حافظه اشاره می‌کند که

حاوی مقدار رشته به اضافه یک نشانگر طول ۳۲-بیت است. جدول زیر آرایش یک بلوک حافظه رشته پهن را در ویندوز نشان می‌دهد.

Table 11.2 آرایش حافظه پویای رشته پهن (تنها برای ویندوز)

آفست	محتویات
–4	نشانگر طول ۳۲-بیت (به بایت)
0..*Length* – 1	رشته کاراکتر
Length	کاراکتر NULL

طول رشته، برابر تعداد بایت‌ها می‌باشد، از این رو دو برابر تعداد کاراکترهای پهن مندرج در رشته است. کاراکتر NULL واقع در انتهای یک بلوک حافظه رشته پهن به طور خودکار توسط کامپایلر و روتین‌های اداره کننده رشته توکار نگه داری و پشتیبانی می‌شود. این امر قالب‌بندی (تبدیل نوع صریح) یک رشته پهن را به طور مستقیم به یک رشته منتهی به تهی امکان‌پذیر می‌سازد.

انواع مجموعه

یک مجموعه، یک آرایه بیتی است جایی که هر بیت نشان می‌دهد که آیا یک عنصر در مجموعه هست یا نه. تعداد ماکزیمم عناصر واقع در یک مجموعه ۲۵۶ است، بنابراین یک مجموعه هرگز بیشتر از ۳۲ بایت را اشغال نمی‌کند. تعداد بایت‌های اشغال شده توسط یک مجموعه به خصوص برابر است با

```
(Max div 8) – (Min div 8) + 1
```

جایی که *Max* و *Min* حدود بالایی و پایینی نوع مبنای مجموعه هستند. تعداد بایت‌های یک عنصر به خصوص E برابر است با

```
(E div 8) – (Min div 8)
```

و تعداد بیت‌های واقع در میان آن بایت برابر است با

```
E mod 8
```

جایی که E مقدار ترتیبی عنصر را مشخص می‌کند. هرگاه امکان‌پذیر باشد، کامپایلر مجموعه‌ها را در ثباتهای CPU ذخیره می‌کند، اما اگر یک مجموعه بزرگتر از نوع *Integer* عمومی‌باشد یا اگر برنامه حاوی کدی باشد که آدرس مجموعه را دریافت می‌کند، مجموعه همواره در حافظه جای می‌گیرد.

انواع آرایه استاتیک

یک آرایه استاتیک به صورت دنباله پیوسته‌ای از متغیرها از نوع عنصر آرایه، ذخیره می‌شود. عناصر با پایین‌ترین اندیس‌ها در پایین‌ترین آدرس‌های حافظه ذخیره می‌شوند. یک آرایه چند بعدی در ابتدا با افزایش سمت راست‌ترین بعد، ذخیره می‌شود.

انواع آرایه دینامیک

یک متغیر آرایه پویا چهار بایت از حافظه را اشغال می‌کند که حاوی یک اشاره‌گر به آرایه تخصیص یافته به طور پویاست. هرگاه متغیر تهی باشد (مقداردهی اولیه نشده باشد) یا آرایه‌ای با طول صفر را نگه دارد، اشاره‌گر **nil** خواهد بود و هیچ حافظه پویایی با متغیر مرتبط نخواهد بود. برای یک آرایه غیرتهی، متغیر به یک بلوک تخصیص یافته به طور پویای حافظه اشاره می‌کند که حاوی آرایه به اضافه یک نشانگر طول ۳۲-بیت و یک شمار ارجاع ۳۲-بیت است. جدول زیر آرایش یک بلوک حافظه آرایه پویا را نشان می‌دهد.

Table 11.3 آرایش حافظه آرایه پویا

آفست	محتویات
-8	شمار ارجاع ۳۲-بیت
-4	نشانگر طول ۳۲-بیت (تعداد عناصر)
0..*Length* * (size of element) – 1	عناصر آرایه

انواع رکورد

هرگاه یک نوع رکورد در حالت **{$A+}** (حالت پیش فرض) اعلان شود و هرگاه اعلان شامل یک تعدیل کننده **packed** نباشد، نوع یک نوع رکورد غیربسته‌ای[۱] می‌باشد، و فیلدهای رکورد برای

[۱] unpacked record type

دسترسی کارآمد توسط CPU، تراز می‌شوند. تراز توسط نوع هر فیلد کنترل می‌شود. هر نوع داده یک تراز ذاتی دارد، که به طور خودکار توسط کامپایلر محاسبه می‌شود. تراز می‌تواند ۱، ۲، ۴ یا ۸ باشد و بیانگر کرانه‌ای است که مقدار یک نوع بایستی در آن ذخیره شود تا کارآمدترین دسترسی را فراهم کند. جدول زیر ترازها را برای تمامی انواع داده‌ها فهرست می‌کند.

ماسک‌های تنظیم نوع **Table 11.4**

نوع	تنظیم
انواع ترتیبی	اندازه نوع (۱، ۲، ۴ یا ۸)
انواع حقیقی	۲ برای Real48، ۴ برای Single، ۸ برای Double و Extended
انواع رشته کوتاه	۱
انواع آرایه	همانی که برای نوع عنصر آرایه است.
انواع رکورد	بزرگترین تنظیم فیلدهای واقع در آرایه
انواع مجموعه	اگر ۱، ۲ یا ۴ باشد، اندازه نوع، در غیر این صورت ۱
تمامی‌انواع دیگر	۴

برای مطمئن شدن از تراز صحیح فیلدهای واقع در یک نوع رکورد غیر بسته‌ای، کامپایلر بایت استفاده نشده‌ای را قبل از فیلدها با یک تراز ۲، درج می‌کند و چنان چه لازم باشد، تا ۳ بایت استفاده نشده را قبل از فیلدها با یک تراز ۴ درج می‌کند. بالاخره کامپایلر اندازه کامل رکورد را تا کران بایت مشخص شده توسط بزرگترین تراز هر یک از فیلدها، گرد می‌کند.

هرگاه یک نوع رکورد در حالت **{$A–}** اعلان شود، یا هرگاه اعلان شامل تعدیل کننده **packed** باشد، فیلدهای رکورد تراز نمی‌شوند، اما در عوض به آفست‌های متوالی تخصیص داده می‌شوند. اندازه کامل یک چنین رکورد بسته‌ای در واقع اندازه همه فیلدها خواهد بود. از آن جایی که تراز داده‌ها می‌تواند تغییر کند، این ایده خوبی است که هر ساختار رکورد را که تمایل دارید به روی دیسک بنویسید یا در حافظه به مدول دیگری که با استفاده از یک نسخه متفاوت کامپایلر، کامپایل شده است، ارسال کنید، بسته‌بندی کنید.

انواع فایل

انواع فایل همانند رکوردها نمایش داده می‌شوند. فایل‌های نوعدار و فایل‌های بدون نوع ۳۳۲ بایـت را اشغال می‌کنند، که همانند زیر آرایش داده می‌شوند:

```
type
TFileRec = packed record
Handle: Integer;
Mode: word;
Flags: word;
case Byte of
0: (RecSize: Cardinal);
1: (BufSize: Cardinal;
BufPos: Cardinal;
BufEnd: Cardinal;
BufPtr: PChar;
OpenFunc: Pointer;
InOutFunc: Pointer;
FlushFunc: Pointer;
CloseFunc: Pointer;
UserData: array[1..32] of Byte;
Name: array[0..259] of Char; );
end;
```

فایل‌های متن ۴۶۰ بایت را اشغال می‌کنند، که همانند زیر آرایش داده می‌شوند:

```
type
TTextBuf = array[0..127] of Char;
TTextRec = packed record
Handle: Integer;
Mode: word;
Flags: word;
BufSize: Cardinal;
BufPos: Cardinal;
BufEnd: Cardinal;
BufPtr: PChar;
OpenFunc: Pointer;
InOutFunc: Pointer;
FlushFunc: Pointer;
CloseFunc: Pointer;
UserData: array[1..32] of Byte;
Name: array[0..259] of Char;
Buffer: TTextBuf;
end;
```

Handle حاوی دستگیره فایل است (هر گاه فایل باز باشد).

فیلد *Mode* می‌تواند یکی از مقادیر زیر را تقبل کند

```
const
fmClosed = $D7B0;
fmInput = $D7B1;
fmOutput = $D7B2;
fmInOut = $D7B3;
```

جایی که *fmClosed* نشان می‌دهد که فایل بسته شده است، *fmInput* و *fmOutput* فایل متنی را نشان می‌دهند که راه‌اندازی شده است (*fmInput*) یا بازنویسی شده است (*fmOutput*)، *fmInOut* یک فایل نوعدار یا بدون نوع را نشان می‌دهد که راه‌اندازی یا بازنویسی شده است. هر مقدار دیگری نشان می‌دهد که متغیر فایل تخصیص داده نشده است (و از این رو مقداردهی اولیه نشده است). فیلد *UserData* برای روتین‌های نوشته شده توسط کاربر در دسترس است تا داده‌ها را ذخیره کند. *Name* حاوی نام فایل است، که دنباله‌ای از کاراکترهاست که توسط یک کاراکتر تهی (#0) خاتمه داده می‌شود. برای فایل‌های نوعدار و فایل‌های بدون نوع، *RecSize* حاوی طول رکورد به بایت است و فیلد *Private* نامستعمل اما رزرو شده است.

برای فایل‌های متن، *BufPtr* یک اشاره‌گر به بافری از بایت‌های *BufSize* است، *BufPos* اندیس کاراکتر بعدی واقع در بافر برای خواندن یا نوشتن است و *BufEnd* یک شمار کاراکترهای معتبر در بافر است. *OpenFunc*، *InOutFunc*، *FlushFunc* یا *CloseFunc* اشاره‌گرهایی به روتین‌های I/O که فایل را کنترل می‌کنند، هستند؛ بخش «توابع ابزار» را در فصل۸ ملاحظه نمایید. Flags سبک سطر جدید را به صورت زیر مشخص می‌کند:

```
bit 0 clear      LF line breaks
bit 0 set        CRLF line breaks
```

تمامی بیت‌های Flags دیگر برای استفاده در آینده رزرو شده اند. *DefaultTextLineBreakStyle* و *SetLineBreakStyle* را هم ملاحظه نمایید.

انواع رویه‌ای

یک اشاره‌گر روال به صورت یک اشاره‌گر ۳۲-بیت به مدخل یک روال یا تابع ذخیره می‌شود. یک اشاره‌گر متد به صورت یک اشاره‌گر ۳۲-بیت به مدخل یک متد ذخیره می‌شود، که توسط یک اشاره‌گر ۳۲-بیت به یک شیء پی گرفته می‌شود.

انواع کلاس

یک مقدار نوع کلاس به صورت یک اشاره‌گر به وهله‌ای از کلاس ذخیره می‌شود، که این وهله، شیء (*object*) نامیده می‌شود. فرمت درونی داده یک شیء شبیه فرمت درونی داده یک رکورد است. فیلدهای شیء به ترتیب اعلان به صورت دنباله‌ای از متغیرهای متوالی ذخیره می‌شوند. فیلدها همواره

مطابق با یک نوع رکورد غیر بسته‌ای، تراز می‌شوند. هر فیلد به ارث رسیده از یک کلاس نیا قبل از فیلدهای جدید تعریف شده در کلاس فرزند، ذخیره می‌شود.

فیلد چهار بایت ابتدایی هر شیء، یک اشاره‌گر به جدول متد مجازی[۱] (VMT) کلاس است. به ازای هر کلاس دقیقاً یک VMT وجود دارد (نه به ازای یک شیء)؛ انواع کلاس مجزا— اهمیتی ندارد که چگونه مشابه اند— هرگز یک VMT را به اشتراک نمی‌گذارند. VMTها به طور خودکار توسط کامپایلر ساخته می‌شوند و هرگز به طور مستقیم توسط یک برنامه دست کاری نمی‌شوند. در ضمن اشاره‌گرها به VMTها، که به طور خودکار توسط متدهای سازنده در اشیائی که آنها ایجاد می‌کنند، ذخیره می‌شوند، هرگز به طور مستقیم توسط یک برنامه دست کاری نمی‌شوند.

آرایش یک VMT در جدول زیر نشان داده شده است. در آفست‌های مثبت، یک VMT از لیستی از اشاره‌گرهای متد ۳۲-بیت— یکی به ازای هر متد مجازی تعریف شده توسط کاربر در نوع کلاس— به ترتیب اعلان، تشکیل می‌یابد. هر شیار حاوی آدرس مرتبط با مدخل متد مجازی است. این آرایش با V-table متعلق به C++ و با COM سازگار است. در آفست‌های منفی، یک VMT حاوی تعدادی از فیلدهاست که برای پیاده‌سازی پاسکال شیئی درونی هستند. از آن جایی که آرایش احتمالاً در پیاده‌سازی‌های آینده پاسکال شیئی تغییر می‌کند، برنامه‌ها بایستی از متدهای تعریف شده در *TObject* برای جستجوی این اطلاعات استفاده کنند.

آرایش جدول متد مجازی (VMT) **Table 11.5**

افست	نوع	توضیح
-76	Pointer	اشاره‌گر به جدول متد مجازی (یا **nil**)
-72	Pointer	اشاره‌گر به جدول واسط (یا **nil**)
-68	Pointer	اشاره‌گر به جدول اطلاعات اتوماسیون (یا **nil**)
-64	Pointer	اشاره‌گر به جدول مقداردهی اولیه وهله (یا **nil**)
-60	Pointer	اشاره‌گر به جدول اطلاعات نوع (یا **nil**)
-56	Pointer	اشاره‌گر به جدول تعریف فیلد (یا **nil**)
-52	Pointer	اشاره‌گر به جدول تعریف متد (یا **nil**)

[۱] Virtual Method Table

-48	Pointer	اشاره‌گر به جدول متد پویا (یا **nil**)
-44	Pointer	اشاره‌گر به رشته کوتاه محتوی نام کلاس
-40	Cardinal	اندازه وهله به بایت
-36	Pointer	اشاره‌گر به یک اشاره‌گر به کلاس نیا (یا **nil**)
-32	Pointer	اشاره‌گر به مدخل متد *SafecallException* (یا **nil**)
-28	Pointer	مدخل متد *AfterConstruction*
-24	Pointer	مدخل متد *BeforeDestruction*
-20	Pointer	مدخل متد *Dispatch*
-16	Pointer	مدخل متد *DefaultHandler*
-12	Pointer	مدخل متد *NewInstance*
-8	Pointer	مدخل متد *FreeInstance*
-4	Pointer	مدخل تخریب کننده *Destroy*
0	Pointer	مدخل اولین متد مجازی تعریف شده توسط کابر
4	Pointer	مدخل دومین متد مجازی تعریف شده توسط کابر
...	...	...

انواع ارجاع کلاس

یک مقدار ارجاع کلاس به صورت یک اشاره‌گر ۳۲-بیت به جدول متد مجازی (VMT) یک کلاس، ذخیره می‌شود.

انواع واریانت

یک واریانت به صورت یک رکورد ۱۶-بایت ذخیره می‌شود که حاوی یک کد نوع و یک مقدار (یا ارجاع به یک مقدار) از نوع داده شده توسط کد است. یونیت‌های *System* و *Variants* ثابت‌ها و انواع را برای واریانت‌ها تعریف می‌کنند.

نوع *TVarData* بیانگر ساختار درونی یک متغیر *Variant* است (در ویندوز، برابر با نوع *Variant* استفاده شده توسط COM و Win32 API است). نوع *TVarData* می‌تواند در قالب‌بندی متغیرهای *Variant* برای دسترسی به ساختار درونی یک متغیر به کار برده شود.

فیلد *VType* یک رکورد *TVarData* حاوی کد نوع واریانت در دوازده بیت پایین‌تر (بیت‌های تعریف شده توسط ثابت *varTypeMask*) است. علاوه بر این، بیت *varArray* ممکن است تنظیم شود تا نشان دهد که واریانت یک آرایه است و بیت *varByRef* ممکن است تنظیم شود تا نشان دهد که واریانت حاوی یک ارجاع در مقابل یک مقدار است. فیلدهای *Reserved1*، *Reserved2* و *Reserved3* یک رکورد *TVarData*، نامستعمل هستند.

محتویات هشت بایت باقیمانده یک رکورد *TVarData* به فیلد *VType* بستگی دارد. اگر هیچ یک از بیت‌های *varArray* و *varByRef* تنظیم نشوند، واریانت حاوی یک مقدار از نوع داده شده خواهد بود. چنان چه بیت *varArray* تنظیم شود، واریانت حاوی یک اشاره‌گر به یک ساختار *TVarArray* خواهد شد که یک آرایه را تعریف می‌کند. نوع هر عنصر آرایه توسط بیت‌های *varTypeMask* واقع در فیلد *VType* داده می‌شود.

اگر بیت *varByRef* تنظیم شود، واریانت حاوی یک ارجاع به مقداری از نوع داده شده توسط بیت‌های *varTypeMask* و *varArray* در فیلد *VType* خواهد شد.

کد نوع *varString* خصوصی (private) است. واریانت‌های حاوی یک مقدار *varString* نبایستی هرگز به یک تابع غیر–دلفی ارسال شوند. در ویندوز، پشتیبان اتوماسیون دلفی به طور خودکار واریانت‌های *varString* را قبل از ارسال آنها به عنوان پارامتر به توابع بیرونی، به واریانت‌های *varOleStr* تبدیل می‌کند.

در لینوکس، *VT_decimal* پشتیبانی نمی‌شود.

فصل ۱۲

کنترل برنامه

این فصل توضیح می‌دهد که پارامترها و نتایج توابع چگونه ذخیره شده و واگذار می‌شوند. بخش نهایی درباره روال‌های خروج بحث می‌کند.

پارامترها و نتایج تابع

طرز عمل پارامترها و نتایج توابع توسط فاکتورهای متعددی از جمله قراردادهای فراخوانی، معناشناختی پارامتر و نوع و اندازه مقدار در حال ارسال شدن، مشخص می‌شود.

ارسال پارامتر

پارامترها، بسته به قرارداد فراخوانی روتین، توسط ثباتهای CPU یا پشته به توابع و روال‌ها واگذار می‌شوند. برای آگاهی از اطلاعات بیشتر درباره قراردادهای فراخوانی، بخش «قراردادهای فراخوانی» را در فصل۶ ملاحظه نمایید.

پارامترهای متغیر (**var**) همواره به واسطه ارجاع (by reference) ارسال می‌شوند، همانند اشاره‌گرهای ۳۲-بیت که به موقعیت واقعی انباره اشاره می‌کنند.

پارامترهای مقدار و ثابت (**const**) بسته به نوع و اندازه پارامتر، به واسطه مقدار (by value) یا به واسطه ارجاع (by reference) ارسال می‌شوند:

- یک پارامتر ترتیبی، با استفاده از فرمت یکسان با یک متغیر از نوع متناظر، به صورت یک مقدار ۸-بیت، ۱۶-بیت، ۳۲-بیت یا ۶۴-بیت ارسال می‌شود.
- یک پارامتر حقیقی همواره روی پشته ارسال می‌شود. یک پارامتر *Single*، ۴ بایت را اشغال می‌کند و یک پارامتر *Double*، *Comp* یا *Currency* ۸ بایت را اشغال می‌کند. یک *Real48*، ۸ بایت را، با مقدار *Real48* ذخیره شده در ۶ بایت پایین‌تر، اشغال می‌کند. یک *Extended*، ۱۲ بایت را، با مقدار *Extended* ذخیره شده در ۱۰ بایت پایین‌تر، اشغال می‌کند.
- یک پارامتر رشته کوتاه به صورت یک اشاره‌گر ۳۲-بیت به یک رشته کوتاه ارسال می‌شود.
- یک پارامتر رشته بلند یا آرایه پویا به عنوان یک اشاره‌گر ۳۲-بیت به بلوک حافظه پویای تخصیص یافته برای رشته بلند، ارسال می‌شود. مقدار **nil** برای یک رشته بلند تهی ارسال می‌شود.
- یک پارامتر اشاره‌گر، کلاس، ارجاع-کلاس یا اشاره‌گر-روال به صورت یک اشاره‌گر ۳۲-بیت ارسال می‌شود.
- یک اشاره‌گر متد روی پشته به صورت دو اشاره‌گر ۳۲-بیت ارسال می‌شود. اشاره‌گر وهله قبل از اشاره‌گر متد به جلو رانده می‌شود به طوری که اشاره‌گر متد پایین‌ترین آدرس را اشغال می‌کند.
- تحت قراردادهای **register** و **pascal**، یک پارامتر واریانت به صورت یک اشاره‌گر ۳۲-بیت به یک مقدار *Variant* ارسال می‌شود.
- مجموعه‌ها، رکوردها و آرایه‌های استاتیک از ۱، ۲ یا ۴ بایت به صورت مقادیر ۸-بیت، ۱۶-بیت و ۳۲-بیت ارسال می‌شوند. مجموعه‌ها، رکوردها و آرایه‌های استاتیک بزرگتر به صورت اشاره‌گرهای ۳۲-بیت به مقدار، ارسال می‌شوند. یک استثنا برای این قاعده، این است که رکوردها همواره به طور مستقیم روی پشته تحت قراردادهای **cdecl**، **stdcall** و **safecall** ارسال می‌شوند؛ اندازه یک رکورد ارسال شده به این روش تا نزدیک ترین کران کلمه-مضاعف گرد می‌شود.
- یک پارامتر آرایه-باز به صورت دو مقدار ۳۲-بیت ارسال می‌شود. اولین مقدار یک اشاره‌گر به داده آرایه، و دومین مقدار یک واحد کمتر از تعداد عناصر واقع در آرایه است.

هرگاه دو پارامتر روی پشته ارسال شوند، هر پارامتر مضربی از ۴ بایت را اشغال می‌کند (یک عدد کامل از کلمات مضاعف). برای یک پارامتر ۸–بیت یا ۱۶–بیت، ولو اینکه پارامتر تنها یک بایت یا یک کلمه را اشغال کند، به صورت یک کلمه مضاعف ارسال می‌شود. محتوای بخش‌های استفاده نشده کلمه مضاعف، تعریف نشده خواهند بود.

تحت قراردادهای **pascal**، **cdecl**، **stdcall** و **safecall**، تمامی‌پارامترها روی پشته ارسال می‌شوند. تحت قرارداد **pascal**، پارامترها به ترتیب اعلانشان (چپ به راست) منتقل (push) می‌شوند، تا این که اولین پارامتر در بالاترین آدرس خاتمه یافته و آخرین پارامتر در پایین‌ترین آدرس خاتمه می‌یابد. تحت قراردادهای **cdecl**، **stdcall** و **safecall**، پارامترها بر خلاف جهت ترتیب اعلان (راست به چپ) منتقل (push) می‌شوند، به طوری که اولین پارامتر در پایین‌ترین آدرس خاتمه یافته و آخرین پارامتر در بالاترین آدرس خاتمه می‌یابد.

تحت قرارداد **register**، تا سه پارامتر در ثباتهای CPU ارسال می‌شوند و پارامترهای الباقی (اگر پارامتری باشد) روی پشته ارسال می‌شوند. پارامترها به ترتیب اعلانشان ارسال می‌شوند (مشابه با قرارداد **pascal**)، و سه پارامتر ابتدایی که واجد شرایط هستند در ثباتهای EAX، EDX و ECX، به همان ترتیب، ارسال می‌شوند. انواع حقیقی، اشاره‌گر متد، واریانت، *Int64* و انواع ساخت یافته واجد شرایط به عنوان پارامترهای ثبات نیستند، اما پارامترهای دیگر هستند. اگر بیشتر از سه پارامتر واجد شرایط پارامترهای ثبات باشند، سه پارامتر ابتدایی در EAX، EDX و ECX ارسال می‌شوند و پارامترهای باقیمانده به ترتیب اعلانشان، به روی پشته منتقل می‌شوند(PUSH). برای مثال، با اعلان‌های داده شده

```
procedure Test(A: Integer; var B: Char; C: Double; const D: string; E: Pointer);
```

یک فراخوان به *Test*، A را در EAX به صورت یک عدد صحیح ۳۲–بیت، B را در EDX به عنوان یک اشاره‌گر به یک *Char*، و D را در ECX به عنوان یک بلوک حافظه رشته بلند ارسال می‌کند؛ C و E به عنوان دو کلمه–مضاعف و یک اشاره‌گر ۳۲–بیت، به همان ترتیب، به روی پشته منتقل می‌شوند.

قراردادهای ذخیره سازی ثبات

روال‌ها و توابع باید ثباتهای EBX، ESI، EDI و EBP را حفظ کنند، اما می‌توانند ثباتهای EAX، EDX و ECX را تغییر دهند. هنگام پیاده‌سازی یک سازنده یا تخریب کننده در اسمبلر، خاطرجمع باشید که

ثبات DL را نگه می‌دارید. روال‌ها و توابع با فرض این که پرچم مسیر CPU پاک شده باشد (متناظر با یک دستورالعمل CLD) احضار می‌شوند و باید با پرچم مسیر ترخیص شده برگردند.

نتایج تابع

قراردادهای زیر برای برگشت دادن مقادیر نتیجه تابع به کار برده می‌شوند.

- هر زمان که امکان‌پذیر باشد، نتایج ترتیبی در یک ثبات CPU برگردانده می‌شوند. بایت‌ها در AL برگردانده می‌شوند، کلمه‌ها در AX و کلمه–مضاعف‌ها در EAX برگردانده می‌شوند.
- نتایج حقیقی در ثبات بالای–پشته کمک پردازنده ممیز شناور ((ST(0)) برگردانده می‌شوند. برای نتایج تابع از نوع *Currency*، مقدار ST(0) توسط ۱۰۰۰۰ مدرج می‌شود. برای مثال، مقدار *Currency* برابر 1.234 در ST(0) به صورت 12340 برگردانده می‌شود.
- برای یک نتیجه رشته، آرایه پویا، اشاره‌گر متد، واریانت یا *Int64*، نتایج مثل همانی است که اگر نتیجه تابع به صورت یک پارامتر **var** اضافی بعد از پارامترهای اعلان شده، اعلان شده بود. به عبارت دیگر، فراخواننده یک اشاره‌گر ۳۲–بیت اضافی ارسال می‌کند که به یک متغیر که در آن نتیجه تابع برمی‌گردد، اشاره می‌کند.
- نتایج اشاره‌گر، کلاس، ارجاع–کلاس، و اشاره‌گر روال در EAX برگردانده می‌شوند.
- برای نتایج آرایه استاتیک، رکورد، و مجموعه، چنان چه مقدار یک بایت را اشغال کند در AL برگردانده می‌شود؛ اگر مقدار دو بایت را اشغال کند در AX برگردانده می‌شود؛ و اگر مقدار چهار بایت را اشغال کند در EAX برگردانده می‌شود. در غیر این صورت، نتیجه در یک پارامتر **var** اضافی که بعد از پارامترهای اعلان شده، به تابع ارسال شده است، برگردانده می‌شود.

فراخوان‌های متد

متدها از همان قراردادهای فراخوانی استفاده می‌کنند که روال‌ها و توابع معمولی آنها را به کار می‌برند، به جز این که هر متد یک پارامتر ضمنی اضافی به نام *Self* دارد، که یک ارجاع به وهله یا کلاسی که در آن متد فراخوان شده است، می‌باشد. پارامتر *Self* به صورت یک اشاره‌گر ۳۲–بیت ارسال می‌شود.

- تحت قرارداد **register**، *Self* طوری رفتار می‌کند که انگار قبل از همه پارامترهای دیگر اعلان شده بود. از این رو *Self* همواره در ثبات EAX ارسال می‌شود.

- تحت قرارداد **pascal**، *Self* طوری رفتار می‌کند که انگار بعد از همه پارامترهای دیگر اعلان شده بود. (شامل پارامتر **var** اضافی که برخی اوقات برای یک نتیجه تابع ارسال می‌شود). از این رو، با خاتمه یافتن در آدرسی پایین‌تر از همه پارامترهای دیگر، آخر از همه منتقل (PUSH) می‌شود.
- تحت قراردادهای **cdecl**، **stdcall** و **safecall** ، *Self* طوری رفتار می‌کند که انگار قبل از همه پارامترهای دیگر، اما بعد از پارامتر **var** اضافی (چنان چه موجود باشد) ارسال شده برای یک نتیجه تابع، اعلان شده بود.

سازنده‌ها و تخریب کننده‌ها

سازنده‌ها و تخریب کننده‌ها همان قراردادهای فراخوانی را استفاده می‌کنند که متدها به کار می‌برند، به جز این که یک پارامتر پرچم *Boolean* اضافی، برای نشان دادن زمینه فراخوانی سازنده یا تخریب کننده، ارسال می‌شود.

یک مقدار *False* در پارامتر پرچم یک فراخوانی سازنده نشان می‌دهد که سازنده از طریق یک وهله شیء یا با استفاده از واژه کلیدی **inherited** احضار شده است. در این حالت، سازنده مانند یک متد معمولی رفتار می‌کند. یک مقدار *True* در پارامتر پرچم یک فراخوانی سازنده نشان می‌دهد که سازنده از طریق یک ارجاع کلاس احضار شده است. در این حالت، سازنده یک وهله از کلاس داده شده توسط *Self* ایجاد می‌کند و یک ارجاع به شیئی که به تازگی ایجاد شده، در EAX برمی‌گرداند.

یک مقدار *False* در پارامتر پرچم یک فراخوانی تخریب کننده نشان می‌دهد که تخریب کننده با استفاده از واژه کلیدی **inherited** احضار شده است. در این حالت، تخریب کننده مانند یک متد معمولی رفتار می‌کند. یک مقدار *True* در پارامتر پرچم یک فراخوانی تخریب کننده نشان می‌دهد که تخریب کننده از طریق یک وهله شیء احضار شده است. در این حالت، تخریب کننده وهله داده شده توسط *Self* را اندکی قبل از برگرداندن آزاد می‌کند.

پارامتر پرچم طوری رفتار می‌کند که انگار قبل از تمامی پارامترها اعلان شده است. تحت قرارداد **register**، این پرچم در ثبات DL ارسال می‌شود. تحت قرارداد **pascal**، پرچم قبل از همه پارامترهای

دیگر منتقل (PUSH) می‌شود. تحت قراردادهای **cdecl**، **stdcall** و **safecall**، پرچم تنها قبل از پارامتر *Self* منتقل (PUSH) می‌شود.

از آن جایی که ثبات DL نشان می‌دهد که آیا سازنده یا تخریب کننده بیرونی‌ترین در پشته فراخوانی است، شما باید مقدار DL را قبل از خروج به حالت اول برگردانید تا *BeforeDestruction* یا *AfterConstruction* بتواند به درستی فراخوان شود.

روال‌های خروج

روال‌های خروج[۱] تضمین می‌کنند که قبل از خاتمه برنامه اعمال به خصوصی— مانند به روزرسانی و بستن فایل‌ها— اجرا می‌شوند. متغیر اشاره‌گر *ExitProc* به شما اجازه می‌دهد تا یک روال خروج را راه‌اندازی کنید، از این رو همواره به عنوان بخشی از پایان برنامه فراخوانده می‌شود— خواه پایان برنامه نرمال باشد، تحمیلی توسط یک فراخوان به *Halt*، یا در نتیجه یک خطای حین اجرا پایان یافته باشد. یک روال خروج هیچ پارامتری نمی‌گیرد.

توجه برای تمامی رفتارهای خروج به جای روال‌های خروج، استفاده از بخش‌های اتمام (finalization) توصیه می‌شود. («بخش اتمام (finalization)» را در فصل ۳ ملاحظه نمایید.) روال‌های خروج تنها برای اهدافی از قبیل فایل‌های اجرایی، اشیای اشتراکی (لینوکس) یا DLL. (ویندوز) در دسترس هستند؛ برای بسته‌ها (packages)، عملکرد خروج باید در بخش اتمام (**finalization**) پیاده‌سازی شود. تمامی‌روال‌های خروج قبل از اجرای بخش‌های اتمام (finalization) فراخوانده می‌شوند.

یونیت‌ها نیز همانند برنامه‌ها می‌توانند روال‌های خروج را نصب کنند. یک یونیت می‌تواند یک روال خروج را، بسته به این که روال فایل‌ها را می‌بندد یا وظایف پاک سازی دیگری را انجام می‌دهد، به عنوان بخشی از کد مقداردهی اولیه (**initialization**) خود نصب کند.

هرگاه روال خروجی به درستی پیاده‌سازی شود، بخشی از یک زنجیره از روال‌های خروج خواهد بود. روال‌ها به ترتیب عکس راه‌اندازی‌شان اجرا می‌شوند، با اطمینان از این که کد خروج اولین یونیت قبل

---

[۱] Exit procedures

از کد خروج هر یونیتی که به آن وابسته است اجرا نخواهد شد. برای نگه داشتن زنجیره به صورت دست نخورده، شما باید محتوای فعلی *ExitProc* را قبل از اشاره آن به آدرس روال خروجی مال خودتان، ذخیره کنید. در ضمن، اولین دستور واقع در روال خروج شما، باید مقدار ذخیره شده *ExitProc* را راه‌اندازی مجدد کند.

کد زیر استخوان‌بندی پیاده‌سازی یک روال خروج را نشان می‌دهد.

```
var
ExitSave: Pointer;
procedure MyExit;
begin
ExitProc := ExitSave;          // always restore old vector first
...
end;
begin
ExitSave := ExitProc;
ExitProc := @MyExit;
...
end.
```

روی مدخل، کد محتوای *ExitProc* را در *ExitSave* ذخیره می‌کند، سپس روال *MyExit* را راه‌اندازی می‌کند. هرگاه به عنوان بخشی از فرایند پایاندهی فراخوان شود، اولین چیزی که *MyExit* انجام می‌دهد، این است که روال خروج قبلی را راه‌اندازی مجدد می‌کند.

روتین انقضاء در کتابخانه زمان اجرا به فراخوانی روال‌های خروج تا زمانی که *ExitProc* برابر **nil** شود، ادامه می‌دهد. برای اجتناب از حلقه‌های نامتناهی، *ExitProc* قبل از هر فراخوانی برابر با **nil** می‌شود، از این رو روال خروج بعدی تنها در صورتی فراخوانده می‌شود که روال خروج جاری آدرسی را به *ExitProc* تخصیص دهد. چنان چه خطایی در یک روال خروج رخ دهد، این روال خروج دوباره فراخوان نمی‌شود.

با بررسی متغیر صحیح *ExitCode* و متغیر اشاره‌گر *ErrorAddr*، یک روال خروج می‌تواند از علت خاتمه برنامه آگاه شود. در حالت خاتمه نرمال برنامه، *ExitCode* برابر صفر بوده و *ErrorAddr* هم برابر **nil** خواهد شد. در حالت خاتمه برنامه از طریق یک فراخوان به *Halt*، *ExitCode* حاوی مقدار ارسال شده به *Halt* و *ErrorAddr* هم برابر **nil** خواهد بود. در حالت خاتمه برنامه به واسطه یک خطای زمان اجرا، *ExitCode* حاوی کد خطا و *ErrorAddr* هم حاوی آدرس دستور نامعتبر خواهد بود.

آخرین روال خروج (روالی که توسط کتابخانه زمان اجرا نصب شده است) فایل‌های *Input* و *Output* را می‌بندد. اگر *ErrorAddr* برابر **nil** باشد، یک پیغام خطای زمان اجرا تولید می‌کند. برای تولید پیغام خطای زمان اجرای سفارشی مال خودتان، یک روال خروج نصب کنید که *ErrorAddr* را بررسی کند و چنان چه *ErrorAddr* برابر **nil** باشد یک پیغام بیرون دهد؛ قبل از برگشت دادن، *ErrorAddr* را برابر **nil** قرار دهید تا این که خطا دوباره توسط روال‌های خروج دیگر گزارش نشود.

همین که کتابخانه زمان اجرا تمامی روال‌های خروج را فراخوان کرده باشد، با ارسال مقدار ذخیره شده در *ExitCode* به عنوان یک کد برگشتی، به سیستم عامل برمی‌گردد.

فصل ۱۳

کد اسمبلر درون خطی

اسمبلر توکار[۱] به شما اجازه می‌دهد تا کد اسمبلر را در میان برنامه‌های پاسکال شیئی بنویسید. این اسمبلر ویژگی‌های زیر را دارد:

- اجازه اسمبلی درون خطی را می‌دهد
- همه دستورالعمل‌های یافت شده در Intel Pentium III، SIMD و AMD Athlon (از جمله 3D Now!) را پشتیبانی می‌کند
- هیچ پشتیبان ماکرویی را فراهم نمی‌کند، اما به اسمبلر خالص اجازه اجرای روال را می‌دهد
- اجازه استفاده از شناسه‌های پاسکال شیئی مانند ثابت‌ها، نوع‌ها و متغیرها را در دستورات اسمبلر می‌دهد.

به عنوان یک جانشین برای اسمبلر توکار، شما می‌توانید به فایل‌های شیء که حاوی روال‌ها و توابع بیرونی هستند، متصل شوید. برای اطلاعات بیشتر، بخش «اتصال به فایل‌های شیئی» را در فصل۶ ملاحظه نمایید.

---

[۱] Built-in Assembler

توجه چنان چه کد اسمبلر بیرونی دارید که می‌خواهید از آن در برنامه خود استفاده کنید، باید توجه داشته باشید که لازم است تا آن را در پاسکال شیئی بازنویسی کنید یا دست کم آن را با استفاده از اسمبلر درون خطی پیاده‌سازی مجدد کنید.

دستور asm

اسمبلر درون خطی از طریق دستورات **asm** قابل دسترسی است؛ دستور **asm** قالب زیر را دارد

```
asm statementList end
```

جایی که *statementList* دنباله‌ای از دستورات اسمبلر است که توسط نقطه ویرگول، کاراکترهای انتهای سطر یا توضیحات پاسکال شیئی از هم جدا می‌شوند. توضیحات واقع در یک دستور **asm** باید به سبک توضیحات پاسکال شیئی باشند. یک نقطه ویرگول نشان نمی‌دهد که باقیمانده سطر یک توضیح است. واژه کلیدی **inline** و راهنمای **assembler** تنها برای سازگاری با گذشته پشتیبانی می‌شوند. آنها اثری بر روی کامپایلر ندارند.

استفاده از ثبات

به طور کلی، قواعد استفاده از ثبات در یک دستور **asm** مشابه با استفاده از آنها در یک تابع یا روال **external** (بیرونی) است. یک دستور **asm** باید ثباتهای EDI، ESI، ESP، EBP و EBX را باقی بگذارد، اما می‌تواند آزادانه ثباتهای EAX، ECX و EDX را اصلاح کند. در ورودی یک دستور **asm**، BP به چارچوب پشته جاری اشاره می‌کند، SP به بالای پشته اشاره می‌کند، SS حاوی آدرس قطعه از (سگمنت) قطعه پشته است و DS حاوی آدرس قطعه سگمنت داده است. به استثنای ESP و EBP، یک دستور **asm** می‌تواند هیچ چیزی را درباره محتویات ثبات روی ورودی دستور تقبل نکند.

ترکیب نوشتاری دستور اسمبلر

ترکیب نحوی نوشتاری یک دستور اسمبلر به صورت زیر است

```
Label: Prefix Opcode Operand1, Operand2
```

جایی که *Label* یک برچسب بوده و *Prefix* یک رمزالعمل[۱] پیشوندی اسمبلر است (کد عملیاتی)، *Opcode* یک رمزالعمل دستور اسمبلر یا یک فرمان است و *Operand* یک عبارت اسمبلر است. *Label* و *Prefix* اختیاری هستند. برخی رمزالعمل‌ها تنها یک عملوند می‌گیرند و برخی هیچ عملوندی نمی‌پذیرند.

توضیحات مابین دستورات اسمبلر مجاز هستند، اما نه در داخل آنها. برای مثال،

```
MOV   AX,1   {Initial value}      { OK }
MOV   CX,100   {Count}            { OK }
MOV   {Initial value}  AX,1;      { Error! }
MOV   CX,   {Count} 100           { Error! }
```

برچسب‌ها

برچسب‌ها در دستورهای اسمبلر توکار همان طور که در پاسکال شیئی به کار برده می‌شوند، استفاده می‌شوند— با نوشتن برچسب و یک ویرگول قبل از یک دستور. برای طول یک برچسب هیچ محدودیتی وجود ندارد. همانند پاسکال شیئی، برچسب‌ها باید در یک بخش اعلان **label** واقع در بلوک حاوی دستور **asm**، اعلان شوند. در این جا یک استثنا برای این قاعده وجود دارد: برچسب‌های محلی (*local labels*).

برچسب‌های محلی، برچسب‌هایی هستند که با یک @ شروع می‌شوند. آنها از یک @ که با یک یا چند حرف، رقم، زیر خط یا @ پی گرفته می‌شود، تشکیل می‌شوند. استفاده از برچسب‌های محلی محدود به دستورات **asm** است و دامنه یک برچسب محلی از واژه کلیدی **asm** تا انتهای دستور **asm** که برچسب را در بردارد، امتداد می‌یابد. لزومی ندارد که یک برچسب محلی اعلان شود.

رمزالعمل‌های دستور

اسمبلر توکار از تمامی رمزالعمل‌های مستند شده Intel به منظور استفاده در برنامه‌های عمومی پشتیبانی می‌کند. توجه کنید که دستورالعمل‌های ممتاز سیستم عامل ممکن است پشتیبانی نشده باشند. به ویژه خانواده‌های دستورالعمل زیر پشتیبانی می‌شوند:

- Pentium family

---

[۱] Opcoce (Operation code)

- Pentium Pro and Pentium II
- Pentium III
- Pentium IV

به علاوه، اسمبلر توکار مجموعه دستورالعمل‌های زیر را هم پشتیبانی می‌کند

- AMD 3DNow! (from the AMD K6 onwards)
- AMD Enhanced 3DNow (from the AMD Athlon onwards)

برای توضیح کاملتر درباره هر دستورالعمل، به مستندات ریزپردازنده خود مراجعه کنید.

برآورد دستورالعمل RET

رمزالعمل دستور RET همواره یک برگشت نزدیک[۱] را تولید می‌کند.

برآورد پرش خودکار

اسمبلر توکار دستورالعمل‌های پرش را با انتخاب خودکار کوتاه‌ترین و در نتیجه کارآمدترین قالب یک دستورالعمل پرش، بهینه‌سازی می‌کند، جز این که طور دیگری هدایت شده باشد. این برآورد پرش خودکار به دستورالعمل پرش غیرشرطی (JMP) اعمال می‌شود، و نیز هنگامی که هدف یک برچسب باشد (نه یک روال یا تابع) به تمامی دستورالعمل‌های پرش شرطی هم اعمال می‌شود.

اگر فاصله تا برچسب هدف 128- تا 127 بایت باشد، برای یک دستورالعمل پرش غیرشرطی (JMP)، اسمبلر یک پرش کوتاه (رمزالعمل یک-بایتی پی گرفته شده با یک جابه‌جایی به اندازه یک-بایت) تولید می‌کند. در غیر این صورت یک پرش نزدیک (رمزالعمل یک-بایتی پی گرفته شده با یک جابه‌جایی به اندازه دو-بایت) تولید می‌کند.

برای یک دستور العمل پرش شرطی، یک پرش کوتاه (رمزالعمل یک-بایت پی گرفته شده با یک جابه‌جایی به اندازه یک-بایت) در صورتی تولید می‌شود که فاصله تا برچسب هدف 128- تا 127 بایت باشد. در غیر این صورت اسمبلر توکار یک پرش کوتاه با شرط معکوس تولید می‌کند، که بالغ بر یک پرش نزدیک تا برچسب هدف پرش می‌کند (جمعاً پنج بایت). برای مثال، دستور اسمبلر

```
JC      Stop
```

[۱] near return

جایی که *Stop* در میان ناحیه یک پرش کوتاه نیست، به یک سلسه کد ماشین که متناظر با کدهای زیر است، تبدیل می‌شود:

```
JNC     Skip
JMP     Stop
Skip:
```

پرش به نقاط مدخل روال‌ها و توابع همواره از نوع پرش نزدیک (یعنی near jump) است.

فرامین اسمبلر

اسمبلر توکار سه فرمان تعریفی اسمبلر را پشتیبانی می‌کند: DB (تعریف بایت)، DW (تعریف کلمه)، DD (تعریف کلمه مضاعف). هر یک از اینها داده‌های متناظر با عملوندهایی را که با ویرگول از هم جدا شده و بعد از فرمان می‌آیند، تولید می‌کنند.

قبل از فراگیری مطالب این بخش به یاد داشته باشید که هر بایت برابر ۸ بیت است و هر کلمه برابر ۲ بایت یا ۱۶ بیت است.

```
1 Byte = 8 Bit
1 Word = 2 Byte = 16 Bit
```

فرمان DB دنباله‌ای از بایت‌ها را تولید می‌کند. هر عملوند می‌تواند یک عبارت ثابت با مقداری واقع در دامنه 128- و 255 یا یک رشته کاراکتری از هر طولی باشد. عبارات ثابت یک بایت کد را تولید می‌کنند و رشته‌ها دنباله‌ای از بایت‌ها را با مقادیر متناظر با کد ASCII هر کاراکتر تولید می‌کنند.

فرمان DW دنباله‌ای از کلمه‌ها را تولید می‌کند. هر عملوند می‌تواند یک عبارت ثابت با مقداری واقع در دامنه 32,768- و 65,535 یا یک عبارت آدرس باشد. برای یک عبارت آدرس، اسمبلر توکار یک اشاره‌گر نزدیک تولید می‌کند— یعنی، کلمه‌ای که حاوی بخش آفست آدرس است.

فرمان DD دنباله‌ای از کلمه‌های مضاعف را تولید می‌کند. هر عملوند می‌تواند یک عبارت ثابت با مقداری واقع در دامنه 2,147,483,648– و 4,294,967,295 یا یک عبارت آدرس باشد. برای یک عبارت آدرس، اسمبلر توکار یک اشاره‌گر دور را تولید می‌کند— یعنی کلمه‌ای که حاوی بخش آفست آدرس است که با یک کلمه که حاوی بخش سگمنت آدرس است، پی گرفته می‌شود.

فرمان DQ یک کلمه چهارتایی را برای مقادیر Int64 تعریف می‌کند.

داده‌های تولید شده توسط DB، DW و DD همواره در سگمنت کد ذخیره می‌شوند، درست مانند کد تولید شده توسط دستورات اسمبلر توکار دیگر. برای تولید داده‌های مقداردهی اولیه شده یا نشده در سگمنت داده، شما بایستی از اعلان‌های **var** یا **const** پاسکال شیئی استفاده کنید. در زیر برخی مثال‌ها از فرامین DB، DW و DD آورده شده‌اند.

```
asm
 DB      0FFH                    { One byte }
 DB      0,99                    { Two bytes }
 DB      'A'                     { Ord('A') }
 DB      'Hello world...',0DH,0AH { String followed by CR/LF }
 DB      12,"string"             { Object Pascal style string }
 DW      0FFFFH                  { One word }
 DW      0,9999                  { Two words }
 DW      'A'                     { Same as DB 'A',0 }
 DW      'BA'                    { Same as DB 'A','B' }
 DW      MyVar                   { Offset of MyVar }
 DW      MyProc                  { Offset of MyProc }
 DD      0FFFFFFFFH              { One double-word }
 DD      0,999999999             { Two double-words }
 DD      'A'                     { Same as DB 'A',0,0,0 }
 DD      'DCBA'                  { Same as DB 'A','B','C','D' }
 DD      MyVar                   { Pointer to MyVar }
 DD      MyProc                  { Pointer to MyProc }
end;
```

در توربو اسمبلر، هرگاه یک شناسه بر یک فرمان DB، DW یا DD مقدم شود، باعث اعلان یک متغیر byte-sized، word-sized یا double-word-sized در موقعیت فرمان می‌شود. برای مثال، توربو اسمبلر اجازه دستورات زیر را می‌دهد:

```
ByteVar   DB    ?
WordVar   DW    ?
IntVar    DD    ?
...
          MOV   AL,ByteVar
          MOV   BX,WordVar
          MOV   ECX,IntVar
```

اسمبلر توکار از یک چنین اعلان‌های متغیری پشتیبانی نمی‌کند. تنها نمونه سمبلی که می‌تواند در یک دستور اسمبلر درون خطی تعریف شود یک برچسب است. تمامی متغیرها باید با استفاده از ترکیب نوشتاری پاسکال شیئی اعلان شوند؛ ساختار قبلی می‌تواند توسط ساختار زیر جایگزین شود

```
var
 ByteVar: Byte;
 WordVar: Word;
 IntVar: Integer;
 ...
asm
 MOV     AL,ByteVar
 MOV     BX,WordVar
```

```
  MOV     ECX,IntVar
end;
```

SMALL و LARGE می‌توانند برای تعیین پهنای یک جابه‌جایی به کار برده شوند:

```
MOV     eax,  [large $1234]
```

این دستورالعمل یک تغییر مکان نرمال با یک جابه‌جایی ۳۲-بیت ($00001234) تولید می‌کند.

```
MOV     eax,  [small $1234]
```

دستورالعمل دوم یک تغییر مکان با یک پیشوند ابطال اندازه آدرس و یک جابه‌جایی ۱۶-بیت ($1234) تولید می‌کند.

SMALL می‌تواند برای ذخیره فضا به کار برده شود. مثال زیر یک ابطال اندازه آدرس و یک آدرس ۲-بایت تولید می‌کند (جمعاً در سه بایت).

```
MOV     eax,  [SMALL 123]
```

در مقابل

```
mov     eax,  [123]
```

که هیچ ابطال اندازه آدرسی تولید نمی‌کند و یک آدرس ۴-بیت (جمعاً در چهار بایت) تولید می‌نماید.

دو فرمان اضافی به کد اسمبلی اجازه می‌دهند تا به متد پویا و مجازی دسترسی پیدا کند: VMTOFFSET و DMTINDEX.

VMTOFFSET آفست را در بایت‌های ورودی جدول اشاره‌گر آرگومان متد مجازی از ابتدی جدول متد مجازی (VMT) بازیابی می‌کند. این دستور نیازمند یک نام کلاس مشخص شده به طور کامل همراه با یک نام متد به عنوان پارامتر است، برای مثال، TExample.VirtualMethod.

DMTINDEX اندیس جدول متد پویای متد پویای ارسال شده را بازیابی می‌کند. در ضمن این فرمان نیازمند یک نام کلاس مشخص شده به طور کامل همراه با یک نام متد به عنوان پارامتر است، برای مثال، TExample.DynamicMethod. برای احضار متد پویا، System.@CallDynaInst را با ثبات (E)SI حاوی مقدار به دست آمده از DMTINDEX فراخوانی کنید.

توجه متدها با فرمان "message"، به صورت متدهای پویا پیاده‌سازی می‌شوند و در ضمن می‌توانند با استفاده از تکنیک DMTINDEX فراخوانده شوند. برای مثال:

```
TMyClass = class
   procedure x;  message  MYMESSAGE;
end;
```

مثال زیر هم از DMTINDEX و هم از VMTOFFSET برای دسترسی به متدهای پویا و مجازی استفاده می‌کند:

```
program Project2;
type
  TExample = class
  procedure DynamicMethod; dynamic;
  procedure VirtualMethod; virtual;
end;
  procedure TExample.DynamicMethod;
begin

end;
  procedure TExample.VirtualMethod;
begin

end;
procedure CallDynamicMethod(e: TExample);
  asm
  // Save ESI register
  PUSH    ESI
  // Instance pointer needs to be in EAX
  MOV     EAX, e
  // DMT entry index needs to be in (E)SI
  MOV     ESI, DMTINDEX TExample.DynamicMethod
  // Now call the method
  CALL    System.@CallDynaInst
  // Restore ESI register
  POP     ESI
end;
procedure CallVirtualMethod(e: TExample);
  asm
   // Instance pointer needs to be in EAX
  MOV    EAX, e
   // Retrieve VMT table entry
  MOV    EDX, [EAX]
   // Now call the method at offset VMTOFFSET
  CALL   DWORD PTR  [EDX + VMTOFFSET TExample.VirtualMethod]
end;
  var
  e: TExample;
begin
  e := TExample.Create;
  try
  CallDynamicMethod(e);
  CallVirtualMethod(e);
  finally
  e.Free;
end;
end.
```

عملوندها

عملوندهای اسمبلر توکار عبارت‌هایی هستند که از ثابت‌ها، ثباتها، علائم (سمبل‌ها) و عملگرها تشکیل می‌شوند. در میان عملوندها، کلمات رزرو شده زیر معانی از پیش تعریف شده‌ای دارند.

Table 13.7 واژه‌های کلیدی اسمبلر توکار

AH	BYTE	DMTINDEX	EDI	HIGH	QWORD	TBYTE
AL	CH	DS	EDX	LARGE	SHL	TYPE
AND	CL	DWORD	EIP	LOW	SHR	VMTOFFSET
AX	CS	DX	ES	MOD	SI	WORD
BH	CX	EAX	ESI	NOT	SMALL	XOR
BL	DH	EBP	ESP	OFFSET	SP	
BP	DI	EBX	FS	OR	SS	
BX	DL	ECX	GS	PTR	ST	

کلمات رزرو شده همواره مقدم بر تمامی شناسه‌های تعریف شده توسط کاربر هستند. برای مثال،

```
var
  Ch:  Char;
  ...
asm
  MOV     CH, 1
end;
```

1 را به درون ثبات CH بارگذاری می‌کند، نه متغیر *Ch*. برای دسترسی به یک سمبل تعریف شده توسط کاربر که نام مشابهی با یک کلمه رزرو شده دارد، باید از عملگر ابطال امپرسند (**&**) استفاده کنید:

```
  MOV     &Ch, 1
```

بهترین کار این است که از به کار بردن شناسه‌های تعریف شده توسط کاربری که اسامی یکسانی با کلمات رزرو شده اسمبلر توکار دارند، اجتناب کنید.

عبارت‌ها

اسمبلر توکار همه عبارات را به صورت مقادیر صحیح ۳۲-بیت ارزیابی می‌کند. اسمبلر توکار از مقادیر ممیز شناور و رشته، به استثنای ثابت‌های رشته‌ای، پشتیبانی نمی‌کند.

عبارات از عناصر عبارت و عملگرها ساخته می‌شوند و هر عبارت یک *کلاس عبارت* و *نوع عبارت* متناظر دارد.

تفاوت میان عبارات پاسکال شیئی و اسمبلر

مهم‌ترین تفاوت میان عبارات پاسکال شیئی و عبارات اسمبلر توکار این است که عبارات اسمبلر باید به یک مقدار ثابت تجزیه شوند—مقداری که می‌تواند در زمان کامپایل محاسبه شود. برای مثال، با اعلان‌های داده شده زیر

```
const
  X = 10;
  Y = 20;
var
  Z:  Integer;
```

دستور زیر، یک دستور معتبر است.

```
asm
   MOV     Z,X+Y
end;
```

از آن جایی که هم X و هم Y هر دو ثابت هستند، عبارت X + Y یک روش سرراست برای نوشتن ثابت 30 است و دستورالعمل منتج شده به سادگی مقدار 30 را به متغیر Z منتقل می‌کند. اما اگر X و Y متغیرهایی باشند که—

```
var
  X, Y: Integer;
```

— اسمبلر توکار نمی‌تواند مقدار X + Y را در زمان کامپایل محاسبه کند. در این حالت، برای انتقال مجموع X و Y به Z می‌توانید دستورات زیر را به کار برید

```
asm
  MOV     EAX,X
  ADD     EAX,Y
  MOV     Z,EAX
end;
```

در یک عبارت پاسکال شیئی، یک ارجاع متغیر **محتوای** متغیر را نشان می‌دهد. اما در یک عبارت اسمبلر، یک ارجاع متغیر **آدرس** متغیر را نشان می‌دهد. در پاسکال شیئی عبارت X + 4 (جایی که X یک متغیر است) به معنای محتوای X به اضافه 4 است، در حالی که برای اسمبلر توکار X + 4 به معنای محتوای کلمه واقع در آدرسی که چهار بایت بالاتر از آدرس X قرار دارد. بنابراین، حتی اگر شما مجاز به نوشتن کد زیر باشید

```
asm
   MOV     EAX,X+4
```

```
end;
```

این کد نمی‌تواند مقدار X به اضافه 4 را در AX بارگذاری کند، این کد مقدار یک کلمه ذخیره شده در چهار بایت بالاتر از X را بارگذاری می‌کند. روش صحیح اضافه کردن 4 به محتوای X به صورت زیر است

```
asm
   MOV     EAX,X
   ADD     EAX,4
end;
```

عناصر عبارت

اجزای یک عبارت **ثابت‌ها**، **رجیسترها** و **سمبل‌ها** هستند.

ثابت‌ها

اسمبلر توکار دو نوع ثابت را پشتیبانی می‌کند: ثابت‌های عددی و ثابت‌های رشته‌ای.

ثابت‌های عددی

ثابت‌های عددی باید مقادیر صحیح باشند و مقادیر آنها نیز باید بین 2,147,483,648– و 4,294,967,295 باشد. به طور پیش فرض، ثابت‌های عددی از نمایش دهدهی استفاده می‌کنند، اما اسمبلر توکار از باینری (مبنای ۲)، اکتال (مبنای ۸) و هگزادسیمال (مبنای ۱۶) نیز پشتیبانی می‌کند. نمایش باینری به واسطه نوشتن یک B بعد از عدد، انتخاب می‌شود. نمایش اکتال (مبنای ۸) با نوشتن یک O بعد از عدد، و نمایش هگزادسیمال (مبنای ۱۶) با نوشتن یک H بعد از عدد یا یک **$** قبل از عدد انتخاب می‌شود.

ثابت‌های عددی باید با یکی از ارقام صفر تا ۹ یا کاراکتر **$** شروع شوند. هرگاه یک ثابت هگزادسیمال را با استفاده از پسوند H بنویسید، چنان چه اولین رقم بامعنی یکی از ارقام A تا F باشد، یک صفر اضافی در جلوی عدد لازم می‌شود. برای مثال، 0BAD4H و $BAD4 ثابت هگزادسیمال هستند، اما BAD4H یک شناسه است زیرا با یک حرف شروع می‌شود.

ثابت‌های رشته

ثابت‌های رشته‌ای باید در میان علایم نقل قول منفرد (' ') یا مضاعف (" ") محصور شوند. دو علامت نقل قول متوالی از نوع یکسان به صورت علایم نقل قول محیطی، تنها به عنوان یک کاراکتر به حساب می‌آیند. در این جا برخی نمونه‌ها از ثابت‌های رشته‌ای آورده شده‌اند:

```
'Z'
'Delphi'
`Linux`
"That's all folks"
' "That' 's all folks," he said. '
'100'
' " '
" ' "
```

در فرامین DB، ثابت‌های رشته‌ای از هر طولی مجاز هستند و باعث تخصیص دنباله‌ای از بایت‌های محتوی مقادیر ASCII کاراکترهای واقع در رشته می‌شوند. در تمامی موارد دیگر، یک ثابت رشته می‌تواند بلندتر از چهار کاراکتر نباشد و یک مقدار عددی را معرفی کند که بتواند در یک عبارت شرکت کند. مقدار عددی یک ثابت رشته‌ای به صورت زیر محاسبه می‌شود

```
Ord(Ch1) + Ord(Ch2) shl 8 + Ord(Ch3) shl 16 + Ord(Ch4) shl 24
```

جایی که *Ch1* سمت راستی‌ترین (آخرین) کاراکتر و *Ch4* سمت چپی‌ترین کاراکتر (اولین) هستند. چنان چه رشته کوتاه‌تر از چهار کاراکتر باشد، فرض می‌شود که سمت چپی‌ترین کاراکترها صفر باشند. جدول زیر برخی ثابت‌های رشته‌ای و مقادیر عددی آنها را نشان می‌دهد.

Table 13.2 نمونه‌هایی از رشته‌ها و مقادیرشان

رشته	مقدار
'a'	00000061H
'ba'	00006261H
'cba'	00636261H
'dcba'	64636261H
'a '	00006120H
' a'	20202061H
'a' * 2	000000E2H

00000020H	'a' - 'A'
FFFFFF9EH	not 'a'

ثباتها

سمبل‌های رزرو شده زیر ثباتهای CPU را معرفی می‌کنند:

ثبات‌های CPU **Table 13.3**

مقاصد عمومی ۳۲-بیت	EAX EBX ECX EDX	اندیس یا اشاره‌گر ۳۲-بیت	ESP EBP ESI EDI
مقاصد عمومی ۱۶-بیت	AX BX CX DX	اندیس یا اشاره‌گر ۱۶-بیت	SP BP SI DI
ثباتهای پایین ۸-بیت	AL BL CL DL	ثباتهای سگمنت ۱۶-بیت	CS DS SS ES
		ثباتهای سگمنت ۳۲-بیت	FS GS
ثباتهای بالای ۸-بیت	AH BH CH DH	پشته ثبات کمک پردازنده	ST

هرگاه یک عملوند فقط از نام یک ثبات تشکیل شده باشد، یک **عملوند ثبات** خوانده می‌شود. تمامی ثباتها می‌توانند به عنوان عملوندهای ثبات به کار برده شوند و برخی ثباتها می‌توانند در زمینه‌های دیگر به کار برده شوند.

ثباتهای پایه (BX و BP) و ثباتهای اندیس (SI و DI) می‌توانند در میان کروشه‌ها نوشته شوند. ترکیب‌های ثبات پایه/اندیس معتبر، [BX]، [BP]، [SI]، [DI]، [BX+SI]، [BX+DI]، [BP+SI] و [BP+DI] هستند. در ضمن شما می‌توانید تمامی ثباتهای ۳۲-بیت را شاخص‌دار نمایید— برای مثال، [EAX+ECX]، [ESP] و [ESP+EAX+5].

ثباتهای سگمنت (ES، CS، SS، DS، FS و GS) پشتیبانی می‌شوند، اما سگمنت‌ها به طور معمول در برنامه‌های ۳۲-بیت سودمند نیستند.

سمبل ST بالایی‌ترین ثبات را روی پشته ثبات ممیز شناور 8087 معرفی می‌کند. هر یک از هشت ثبات ممیز شناور می‌توانند به استفاده از ST(*X*) منتسب شوند، جایی که *X* یک ثابت بین صفر و 7 است و نشانگر فاصله از بالای پشته ثبات است.

سمبل‌ها

اسمبلر توکار به شما اجازه می‌دهد تا تقریباً به تمامی شناسه‌های پاسکال شیئی در عبارات اسمبلر دسترسی داشته باشید؛ این شناسه‌ها شامل ثابت‌ها، نوع‌ها، متغیرها، روال‌ها و توابع هستند. علاوه بر این، اسمبلر توکار سمبل ویژه *@Result* را پیاده‌سازی می‌کند، که متناظر با متغیر *Result* در میان بدنه یک تابع است. برای مثال، تابع

```
function Sum(X, Y: Integer): Integer;
begin
  Result := X + Y;
end;
```

می‌تواند در اسمبلر به این صورت نوشته شود

```
function Sum(X, Y: Integer): Integer; stdcall;
begin
asm
  MOV     EAX,X
  ADD     EAX,Y
  MOV     @Result,EAX
end;
end;
```

سمبل‌های زیر نمی‌توانند در دستورات **asm** به کار برده شوند:

- روال‌ها و توابع استاندارد (برای مثال، *WriteLn* و *Chr*).
- ثابت‌های رشته، ممیز شناور و مجموعه (به جز هنگام بارگذاری ثباتها).
- برچسب‌هایی که در بلوک جاری اعلان نشده باشند.
- سمبل *@Result* خارج از توابع.

جدول زیر انواع سمبلی را که می‌توانند در دستورات **asm** به کار برده شوند، جمع‌بندی می‌کند.

Table 13.4 سمبل‌های تأیید شده توسط اسمبلر توکار

سمبل		مقدار	کلاس	نوع
برچسب	(Label)	آدرس برچسب	ارجاع حافظه	اندازه نوع
ثابت	(Constant)	مقدار ثابت	مقدار بی واسطه	صفر
نوع	(Type)	صفر	ارجاع حافظه	اندازه نوع
فیلد	(Field)	آفست فیلد	حافظه	اندازه نوع
متغیر	(Variable)	آدرس متغیر	ارجاع حافظه	اندازه نوع
روال	(Procedure)	آدرس روال	ارجاع حافظه	اندازه نوع
تابع	(Function)	آدرس تابع	ارجاع حافظه	اندازه نوع

یونیت	(Unit) صفر	مقدار بی واسطه	صفر	
@Result	آدرس متغیر Result	ارجاع حافظه	اندازه نوع	

با بهینه سازی‌های غیرفعال شده، متغیرهای محلی (متغیرهای اعلان شده در روال‌ها و توابع) همواره روی پشته تخصیص داده می‌شوند و نسبت به EBP در دسترس می‌باشند و مقدار یک سمبل متغیر محلی برابر آفست علامت دارش از EBP است. اسمبلر به طور خودکار [EBP] را در ارجاعات به متغیرهای محلی اضافه می‌کند. برای مثال، با اعلان‌های داده شده زیر

```
var Count: Integer;
```

در میان یک تابع یا روال، دستورالعمل

```
MOV     EAX,Count
```

به MOV EAX,[EBP–4] اسمبل و برگردانده می‌شود.

اسمبلر توکار با پارامترهای **var** همانند یک اشاره‌گر ۳۲- بیت برخورد می‌کند و اندازه یک پارامتر **var** همواره چهار است. ترکیب نحوی نوشتاری برای دسترسی به یک پارامتر **var** متفاوت از ترکیب نحوی نوشتاری برای دسترسی به یک پارامتر مقدار است. برای دسترسی به محتویات یک پارامتر **var**، شما باید در ابتدا اشاره‌گر ۳۲-بیت را بارگذاری کرده و سپس به موقعیتی که اشاره‌گر به ان اشاره می‌کند دسترسی پیدا کنید. برای مثال،

```
function Sum(var X, Y: Integer): Integer; stdcall;
begin
asm
  MOV     EAX,X
  MOV     EAX,[EAX]
  MOV     EDX,Y
  ADD     EAX,[EDX]
  MOV     @Result,EAX
end;
end;
```

شناسه‌ها می‌توانند در میان دستورات **asm** قیددار شوند. برای مثال، با اعلان‌های داده شده زیر

```
type
  TPoint = record
  X, Y: Integer;
end;
  TRect = record
  A, B: TPoint;
end;
var
```

```
  P: TPoint;
  R: TRect;
```

ساختارهای زیر می‌توانند در یک دستور **asm** برای دسترسی به فیلدها به کار برده شوند.

```
  MOV     EAX,P.X
  MOV     EDX,P.Y
  MOV     ECX,R.A.X
  MOV     EBX,R.B.Y
```

یک شناسه نوع می‌تواند برای ایجاد متغیرهای در حال جنبش به کار برده شود. هر یک از دستورالعمل‌های زیر کد ماشین یکسانی تولید می‌کنند، که این کد محتوای [EDX] را در میان EAX بارگذاری می‌کند.

```
  MOV     EAX,(TRect PTR [EDX]).B.X
  MOV     EAX,TRect([EDX]).B.X
  MOV     EAX,TRect[EDX].B.X
  MOV     EAX,[EDX].TRect.B.X
```

کلاس‌هاس عبارت

اسمبلر توکار عبارات را به سه کلاس تقسیم می‌کند : **ثباتها، ارجاعات حافظه و مقادیر بی‌واسطه**.

یک عبارت که منحصراً از نام ثبات تشکیل شده است، یک عبارت ثبات است. AX، CL، DI و ES نمونه‌هایی از عبارات ثبات هستند. همان طور که برای عملوندها استفاده می‌شود، عبارات ثبات اسمبلر را برای تولید دستورالعمل‌هایی که روی ثباتهای CPU عمل می‌کنند، راهبری می‌کنند.

عباراتی که موقعیت‌های حافظه را مشخص می‌کنند، ارجاعات حافظه هستند. برچسب‌ها، متغیرها، ثابت‌های نوعدار، روال‌ها و توابع پاسکال شیئی به این فهرست تعلق دارند. عباراتی که ثبات نیستند و با موقعیت‌های حافظه متناظر نیستند، مقادیر بی واسطه هستند. این گروه شامل ثابت‌های بدون نوع و شناسه‌های نوع پاسکال شیئی است. مقادیر بی واسطه و ارجاعات حافظه هنگامی که به عنوان عملوند به کار می‌روند، باعث می‌شوند که کد متفاوتی تولید گردد. برای مثال،

```
const
  Start = 10;
var
  Count: Integer;
  ...
asm
  MOV     EAX,Start      { MOV EAX,xxxx }
  MOV     EBX,Count      { MOV EBX,[xxxx] }
  MOV     ECX,[Start]    { MOV ECX,[xxxx] }
```

```
  MOV     EDX,OFFSET Count     { MOV EDX,xxxx }
end;
```

از آن جایی که *Start* یک مقدار بی واسطه است، اولین MOV به یک دستورالعمل جابه‌جایی فوری اسمبل می‌شود. گرچه دومین MOV به یک دستورالعمل تغییر مکان حافظه ترجمه می‌شود، در نتیجه *Count* یک ارجاع حافظه است. در سومین MOV، براکت‌ها *Start* را به یک ارجاع حافظه تبدیل می‌کنند (در این حالت، کلمه واقع در آفست 10 در سگمنت داده). در چهارمین MOV، عملگر OFFSET، *Count* را به یک مقدار بی واسطه تبدیل می‌کند (آفست *Count* در سگمنت داده).

براکت‌ها و عملگر OFFSET مکمل یکدیگر هستند. دستور **asm** زیر کد ماشینی همسان با دو سطر اول دستور **asm** قبلی، تولید می‌کند.

```
asm
  MOV     EAX,OFFSET [Start]
  MOV     EBX,[OFFSET Count]
end;
```

ارجاعات حافظه و مقادیر بی واسطه بیشتر، یا به عنوان **جابه‌جایی‌پذیر** (relocatable) یا به عنوان **مطلق** (absolute) طبقه‌بندی می‌شوند. جابه‌جاسازی فرایندی است که توسط آن پیونددهنده آدرس‌های مطلق را به سمبل‌ها تخصیص می‌دهد. یک عبارت جابه‌جایی‌پذیر مقداری را مشخص می‌کند که نیازمند جابه‌جایی در زمان اتصال است، در حالی که یک عبارت مطلق مقداری را معرفی می‌کند که ابداً به چنان جابه‌جایی نیاز ندارد. برای نمونه، عباراتی که به برچسب‌ها، متغیرها، روال‌ها یا توابع اشاره می‌کنند جابه‌جایی‌پذیر هستند، زیرا آدرس نهایی این سمبل‌ها در زمان کامپایل ناشناخته است. عباراتی که منحصراً روی ثابت‌ها عمل می‌کنند، مطلق هستند.

اسمبلر توکار به شما اجازه می‌دهد تا هر عملیاتی را روی یک مقدار مطلق انجام دهید، اما عملیات‌های اعمالی روی مقادیر جابه‌جایی‌پذیر را به جمع و تفریق ثابت‌ها محدود می‌کند.

انواع عبارت

هر عبارت اسمبلر توکار یک نوع دارد— یا، اگر دقیق‌تر بگوییم، یک اندازه دارد، زیرا اسمبلر نوع یک عبارت را در اصل به صورت اندازه موقعیت حافظه آن در نظر می‌گیرد. برای مثال، نوع یک متغیر *Integer*، چهار است، زیرا *Integer* چهار بایت را اشغال می‌کند. اسمبلر توکار هر زمان که امکان‌پذیر باشد، بررسی نوع را انجام می‌دهد، از این رو در دستورالعمل‌های

```
var
  QuitFlag: Boolean;
  OutBufPtr: Word;
  ...
asm
  MOV     AL,QuitFlag
  MOV     BX,OutBufPtr
end;
```

اسمبلر بررسی می‌کند که اندازه *QuitFlag* یک (بایت) باشد، و این که اندازه *OutBufPtr* دو (دو بایت یا یک کلمه) باشد. دستورالعمل

```
  MOV     DL,OutBufPtr
```

یک خطا تولید می‌کند زیرا DL یک ثبات دارای اندازه بایت است و *OutBufPtr* یک کلمه است. نوع یک ارجاع حافظه می‌تواند از طریق یک قالب‌بندی (تبدل نوع صریح) تغییر داده شود؛ اینها شیوه‌های صحیح نوشتن دستورالعمل قبلی هستند:

```
  MOV     DL,BYTE PTR OutBufPtr
  MOV     DL,Byte(OutBufPtr)
  MOV     DL,OutBufPtr.Byte
```

این دستورالعمل‌های MOV همگی به اولین بایت (کم اهمیت‌ترین بایت) متغیر *OutBufPtr* اشاره می‌کنند. در برخی موارد، یک ارجاع حافظه بدون نوع است. یک مثال، مقدار بی واسطه محصور شده‌ای در میان کروشه‌ها—[ ]— می‌باشد:

```
  MOV     al,   [Buffer]
  MOV     cx,   [Buffer]
  MOV     edx, [Buffer]
```

اسمبلر توکار اجازه هر دو دستورالعمل را می‌دهد، زیرا عبارت [100H] هیچ نوعی ندارد— این عبارت تنها به معنای «محتوای آدرس 100H در سگمنت داده» است و نوع می‌تواند از اولین عملوند مشخص شود (بایت برای AL، کلمه برای BX). در حالاتی که نوع نمی‌تواند از عملوند دیگر تعیین شود، اسمبلر توکار نیاز به یک قالب‌بندی ضمنی دارد:

```
  INC     BYTE   PTR  [ECX]
  IMUL    WORD  PTR  [EDX]
```

جدول زیر سمبل‌های نوع از پیش تعریف شده را جمع‌بندی می‌کند که اسمبلر توکار علاوه بر هر نـوعی که به طور جاری توسط پاسکال شیئی اعلان شده است، آنها را ارایه می‌دهد.

سمبل‌های نوع از پیش تعریف شده **Table 13.5**

نوع	سمبل
1	BYTE
2	WORD
4	DWORD
8	QWORD
10	TBYTE

عملگرهای عبارت

اسمبلر توکار عملگرهای متنوعی را فراهم آورده است. قواعد حق تقدم برای این عملگرها متفـاوت از قواعد حق تقدم پاسکال شیئی هستند. برای مثال، در یک دستور **asm**، عملگر AND تقدم پایین‌تـری از عملگرهای جمع و تفریق دارد. جدول زیر عملگرهای عبارت متعلق به اسمبلر توکـار را بـه ترتیـب کاهش حق تقدم آنها فهرست‌بندی می‌کند.

حق تقدم عملگرهای مربوط به عبارت اسمبلر توکار **Table 13.6**

حق تقدم	نکات	عملگرها
بالاترین		**&**
		(), [], ., HIGH, LOW
	+ و – یکانی unary **+** and **–**	+, –
		:
		OFFSET, TYPE, PTR, *, /, MOD, SHL,
	+ و – باینری binary **+** and **–**	**SHR**, +, –
پایین‌ترین		**NOT, AND, OR, XOR**

جدول زیر عملگرهای عبارت متعلق به اسمبلر توکار را تعریف می‌کند.

تعاریف عملگرهای عبارت اسمبلر توکار **Table 13.7**

عملگر	توضیح
&	**ابطال وتعریف مجدد شناسه.** با شناسه‌ای که بلافاصله بعد از امپرسند می‌آید، به عنوان یک سمبل تعریف شده کاربر رفتار می‌شود، حتی اگر املایش با سمبل رزرو شده اسمبلر توکار یکسان باشد.
(...)	**زیر عبارت.** عبارت‌های واقع در میان پارانتزها قبل از این که با آنها همانند یک جزء عبارت واحد رفتار شود به طور کامل ارزیابی می‌شوند. در میان پارانتزها عبارت دیگری می‌تواند مقدم بر عبارت باشد؛ نتیجه در این حالت برابر مجموع مقادیر دو عبارت با نوع اولین عبارت خواهد بود.
[...]	**ارجاع حافظه.** عبارت واقع در میان براکت‌ها قبل از این که با آنها به صورت یک جزء عبارت واحد رفتار شود، به طور کامل ارزیابی می‌شوند. در داخل براکت‌ها عبارت دیگری می‌تواند مقدم بر عبارت گردد؛ نتیجه در این حالت مجموع مقادیر دو عبارت با نوع مقدار اولی است. نتیجه همواره یک ارجاع حافظه است.
.	**انتخاب کننده عضو ساختار.** نتیجه برابر مجموع عبارت قبل از نقطه و عبارت بعد از نقطه با نوع عبارت بعد از نقطه است. سمبل‌های متعلق به دامنه مشخص شده توسط عبارت قبل از نقطه می‌تواند در عبارت بعد از نقطه قابل دسترس باشد.
HIGH	هشت بیت رتبه بالای عبارت با اندازه کلمه‌ای را که بعد از عملگر می‌آید، برمی‌گرداند. عبارت باید یک مقدار بی واسطه مطلق باشد.
LOW	هشت بیت رتبه پایین عبارت با اندازه کلمه را که بعد از عملگر می‌آید، برمی‌گرداند. عبارت باید یک مقدار بی واسطه مطلق باشد.
+	**جمع یکانی.** عبارت بعد از علامت جمع را بدون تغییر برمی‌گرداند. عبارت باید یک مقدار بی‌واسطه مطلق باشد.
−	**منهای یکانی.** مقدار منفی شده (قرینه) عبارت بعد از علامت منها را برمی‌گرداند. عبارت باید یک مقدار بی واسطه مطلق باشد.
+	**جمع.** عبارات می‌توانند مقادیر بی واسطه یا ارجاعات حافظه باشند، اما تنها یکی از عبارات می‌تواند یک مقدار جابه‌جایی‌پذیر باشد. اگر یکی از عبارات یک مقدار جابه‌جایی‌پذیر باشد، نتیجه نیز یک مقدار جابه‌جایی‌پذیر خواهد بود. اگر هر دو عبارت یک ارجاع حافظه باشند، نتیجه نیز یک ارجاع حافظه خواهد بود.

–	**تفریق.** اولین عبارت می‌تواند هر کلاسی داشته باشد، اما دومین عبارت باید یک مقدار بی واسطه مطلق باشد. نتیجه همان کلاسی را خواهد داشت که اولین عبارت دارد.
:	**ابطال سگمنت.** به اسمبلر دستور می‌دهد که عبارت بعد از ویرگول به سگمنت داده شده توسط اسم ثبات سگمنت (CS، DS، SS، FS، GS یا ES) قبل از ویرگول تعلق دارد. نتیجه یک ارجاع حافظه با مقدار عبارت بعد از ویرگول است. هرگاه یک ابطال سگمنت در یک عملوند دستورالعمل به کار برده شود، دستورالعمل با یک دستورالعمل پیشوندی ابطال سگمنت مناسب، پیشونددار می‌شود تا مطمئن شود که سگمنت نشان داده شده انتخاب شده است.
OFFSET	بخش آفست (کلمه مضاعف) یک عبارت را که بعد از عملگر می‌آید برمی‌گرداند. نتیجه یک مقدار بی‌واسطه است.
TYPE	نوع (اندازه به بایت) عبارتی را که بعد از عملگر می‌آید برمی‌گرداند. نوع یک مقدار بی‌واسطه صفر است.
PTR	**عملگر قالب‌بندی.** نتیجه یک ارجاع حافظه با مقدار عبارتی است که بعد از عملگر می‌آید و نوع عبارتی که در جلوی عملگر قرار دارد.
*	**ضرب.** هر دو عبارت باید مقادیر بی واسطه مطلق باشند و نتیجه یک مقدار بی واسطه مطلق خواهد بود.
/	**تقسیم صحیح.** هر دو عبارت باید مقادیر بی واسطه مطلق باشند و نتیجه یک مقدار بی‌واسطه مطلق خواهد بود.
MOD	**باقیمانده بعد از تقسیم صحیح.** هر دو عبارت باید مقادیر بی واسطه مطلق باشند و نتیجه یک مقدار بی‌واسطه مطلق است.
SHL	**تغییر مکان منطقی به راست.** هر دو عبارت باید مقادیر بی واسطه مطلق باشند و نتیجه یک مقدار بی‌واسطه مطلق است.
SHR	**تغییر مکان منطقی به راست.** هر دو عبارت باید مقادیر بی واسطه مطلق باشند و نتیجه یک مقدار بی‌واسطه مطلق است.
NOT	**نفی بیتی.** عبارت باید مقادیر بی واسطه مطلق باشد و نتیجه یک مقدار بی واسطه مطلق است.
AND	**AND بیتی.** هر دو عبارت باید مقادیر بی واسطه مطلق باشند و نتیجه یک مقدار بی واسطه مطلق است.
OR	**OR بیتی.** هر دو عبارت باید مقادیر بی واسطه مطلق باشند و نتیجه یک مقدار بی واسطه مطلق است.
XOR	**OR انحصاری بیتی.** هر دو عبارت باید مقادیر بی واسطه مطلق باشند و نتیجه یک مقدار بی‌واسطه مطلق است.

توابع و روال‌های اسمبلر

شما می‌توانید بدون به کار بردن دستور **begin...end**، با استفاده از کد اسمبلر درون خطی، روال‌ها و توابع کاملی را بنویسید. برای مثال،

```
function LongMul(X, Y: Integer): Longint;
asm
  MOV     EAX,X
  IMUL    Y
end;
```

کامپایلر بهینه‌سازی‌های متعددی را روی این روتین‌ها انجام می‌دهد:

- هیچ کدی برای کپی پارامترهای مقدار به درون متغیرهای محلی تولید نمی‌شود. این امر بر روی تمامی پارامترهای مقدار نوع رشته و پارامترهای مقدار دیگر که اندازه آنها برابر با ۱، ۲ یا ۴ بایت نیست، اثر می‌گذارد. در میان روتین‌ها، باید با یک چنین پارامترهایی طوری برخورد شود که انگار آنها پارامترهای **var** بوده‌اند.
- کامپایلر یک متغیر نتیجه تابع را تخصیص نمی‌دهد، مگر این که تابع یک رشته، واریانت یا ارجاع واسط را برگرداند؛ یک ارجاع به سمبل *@Result* خطا محسوب می‌شود. برای رشته‌ها، واریانت‌ها و واسط‌ها فراخواننده همواره یک اشاره‌گر *@Result* را تخصیص می‌دهد.
- کامپایلر چارچوب‌های پشته را تنها برای روتین‌های تودرتو، روتین‌هایی که پارامترهای محلی دارند، یا برای روتین‌هایی که پارامترهایی روی پشته دارند، تولید می‌کند.
- کد ورودی و خروجی که به طور خودکار برای روتین تولید می‌شود، به این صورت به نظر می‌آید:

```
  PUSH    EBP ;Present if Locals <> 0 or Params <> 0
  MOV     EBP,ESP ;Present if Locals <> 0 or Params <> 0
  SUB     ESP,Locals ;Present if Locals <> 0
  ...
  MOV     ESP,EBP ;Present if Locals <> 0
  POP     EBP ;Present if Locals <> 0 or Params <> 0
  RET     Params ;Always present
```

اگر *Locals* شامل واریانت‌ها، رشته‌های بلند یا واسط‌ها باشد، با صفر مقداردهی اولیه می‌شوند اما تکمیل نمی‌شوند.

- *Locals* اندازه متغیرهای محلی بوده و *Params* انـدازه پارامترهاسـت. اگـر هـم *Locals* و هـم *Params* صـفر باشـند، کـد ورودی وجـود نخواهـد داشـت، و کـد خـروج در اصـل از یـک دستورالعمل RET تشکیل می‌شود.

توابع اسمبلر نتایج خود را به صورتی که می‌آید برمی‌گردانند.

- مقادیر ترتیبی در AL (مقادیر ۸-بیـت) ، AX (مقـادیر ۱۶-بیـت) یـا EAX (مقـادیر ۳۲-بیـت) برگشت داده می‌شوند.
- مقادیر حقیقی در ST(0) روی پشته رجیستر کمک پردازنده برگشـت داده مـی‌شـوند. (مقـادیر *Currency* با 10000 مدرج می‌شوند.)
- اشاره‌گرها، از جمله رشته‌های بلند، در EAX برگشت داده می‌شوند.
- رشته‌های کوتاه و واریانت‌ها در موقعیت موقتی اشـاره شـده توسـط *@Result* برگشـت داده می‌شوند.

A

ضمیمه

گرامر پاسکال شیئی