

APRENDA PASCAL

Contenido

- [Introducción](#)
- [Compiladores de Pascal](#)
- [Historia de Pascal](#)
- [Hola, mundo](#)
- Fundamentos
 - [Estructura de un programa](#)
 - [Identificadores](#)
 - [Constantes](#)
 - [Variables y tipos de datos](#)
 - [Asignaciones y operaciones](#)
 - [Funciones standard](#)
 - [Puntuación e indentación](#)
 - [Ejemplo N°1](#)
 - [Solución](#)
- Entrada/Salida
 - [Entrada](#)
 - [Salida](#)
 - [Formato de salida](#)
 - [Archivos](#)
 - [EOLN y EOF](#)
 - [Ejemplo N°2](#)
 - [Solución](#)
- Flujo del programa
 - [Control secuencial](#)
 - [Expresiones booleanas](#)
 - Ramificaciones
 - [If](#)
 - [Case](#)
 - Lazos
 - [For do](#)
 - [While do](#)
 - [Repeat until](#)
 - [Ejemplo N°3: Serie de Fibonacci y potencias de dos](#)
 - [Soluciones](#)
- Subrutinas
 - [Procedimientos](#)
 - [Parámetros](#)
 - [Funciones](#)
 - [Scope](#)
 - [Recursividad](#)
 - [Forward](#)
 - [Ejemplo N°4: Las Torres de Hanoi](#)
 - [Solución](#)
- Tipos de datos
 - [Tipos de datos numerados](#)
 - [Subrangos](#)
 - [Arreglos unidimensionales](#)
 - [Arreglos multidimensionales](#)
 - [Estructuras](#)
 - [Punteros](#)
- [Palabras Finales](#)

APRENDA PASCAL

Introducción

Bienvenido a Aprenda Pascal! Este manual es una introducción simple y completa al lenguaje de programación Pascal, incluyendo punteros. Puede encontrar su versión original en :
`http://web.mit.edu/taoyue/www/tutorials/pascal/pas-x.html`.

He tratado de hacer las cosas tan claras como sea posible. Si usted no entiende nada, pruebe los programas en su compilador de Pascal e irá aprendiendo poco a poco algunas cosas. Pascal es un lenguaje sintácticamente estricto. Esto significa que si usted comete un error, el compilador se detendrá y le informará de su error. Si usted quiere ver con mayor profundidad temas que estén mas allá de estructuras de datos básicos, recurra a los libros de la bibliografía de la materia.

Para programar en Pascal (o en cualquier lenguaje de alto nivel), usted necesitará un compilador. Usted puede utilizar cualquier compilador de Pascal standard con este manual. Si necesita descargar un compilador, o si usted es la primera vez que va a programar y no sabe que compilador utilizar visite el link [“Compiladores”](#) para más información.

Por favor elija su destino desde la barra de herramientas de abajo.



[Continuar](#)



[Contenido](#)

APRENDA PASCAL

Compiladores de Pascal

Este documento explicará los conceptos básicos acerca de compiladores así como proveerá links hacia los compiladores de Pascal más conocidos. Si usted está familiarizado con lo que son los compiladores, puede descargarlos en <http://www.freepascal.org/download.html>.

Acerca de Lenguajes de Computación y Compiladores

Cuando hablamos acerca de lenguajes de computación, básicamente los tres términos más importantes que usamos son:

Lenguaje de máquina – código binario que da las instrucciones básicas a la CPU de la computadora. Normalmente son comandos muy simples como sumar DOS números o mover un dato desde una posición de memoria hacia otra.

1. **Lenguaje Ensamblador** -- un medio por el que las personas pueden programar computadoras directamente sin memorizar cadenas de números binarios. Hay una correspondencia uno a uno con el código de máquina. Por ejemplo, en el lenguaje de máquina de los procesadores Intel x86, ADD and MOV son los mnemónicos de las operaciones sumar y mover.
2. **Lenguaje de alto nivel** -- permite a las personas escribir programas complejos sin tener que ir paso a paso. Los lenguajes de alto nivel son Pascal, C, C++, Fortran, Java, Basic, y muchos más. Un comando en lenguaje de alto nivel, como escribir una cadena de caracteres en un archivo, puede traducirse a docenas y aún cientos de instrucciones de lenguaje de máquina.

Los procesadores sólo pueden ejecutar directamente los programas en lenguaje de máquina. Los programas en lenguaje ensamblador son *ensamblados*, o traducidos en lenguaje de máquina. Del mismo modo, los programas escritos en lenguajes de alto nivel, como Pascal, también deben ser traducidos a lenguaje de máquina antes de que puedan ser ejecutados. La terminología técnica para esta operación es **compilar** (también conocida como compilación).

El programa que lleva a cabo la compilación se llama **compilador**. Este programa es más bien complejo ya que no sólo crea instrucciones en lenguaje de máquina a partir de líneas de código, sino que también optimiza el código para ejecutarlo más rápido, incluye corrección de errores de código, y vincula el código con subrutinas almacenadas en otros sitios. Por ejemplo, cuando usted le dice a la computadora que imprima algo en la pantalla, el compilador traduce esto como una llamada a un módulo de impresión previamente escrito. Luego de vincular el código que usted escribió con el código provisto por el fabricante del compilador, resulta un programa ejecutable (.exe).

Con lenguajes de alto nivel, hay nuevamente tres términos para recordar:

1. **Código fuente** – el código que *usted* escribe. Este normalmente tiene una extensión que indica que lenguaje fue utilizado. Por ejemplo, el código fuente de Pascal normalmente es un archivo con extensión “.pas” y el código fuente de C++ normalmente termina con “.cpp”
2. **Código objeto** – el que resulta de compilar. El código objeto normalmente incluye sólo un módulo de un programa, y aún no puede ser ejecutado ya que está incompleto. En sistemas operativos DOS o Windows, el código objeto tiene una extensión “.obj”.
3. **Código ejecutable** – el resultado final. Vincula todos los módulos de código objeto necesarios para que el programa funcione. En sistemas operativos DOS o Windows, tiene normalmente una extensión “.exe”

Más Acerca de Compiladores

El estándar en compiladores basado en DOS o en Windows de hecho es el Borland Pascal. Antes de que éste existiera, la mayoría de los compiladores de Pascal eran poco operativos y lentos, se apartaban del Pascal estándar, y costaban varios cientos de dólares. En 1984, Borland introdujo el Turbo Pascal, que costaba menos de u\$s 100, compilando un orden de magnitud más rápido que los compiladores existentes, con abundante código fuente y programas utilitarios.

Este producto fue un éxito inmediatamente. Sin embargo, en 1993, apareció la última versión de [Turbo Pascal](#), la versión 7 para DOS. Luego de eso, la demanda de programas en DOS cayó en picada y Borland (entonces renombrada Inprise) se enfocó en producir compiladores para Windows.

Este manual sólo trata con programación basada en computadoras, las cuales imprimen líneas de datos en la pantalla y el usuario interactúa con el programa usando un teclado. El propósito de este manual es enseñar *cómo* programar en Pascal. Una vez que usted haya aprendido eso, podrá fácilmente leer los libros de la bibliografía de la materia o páginas web y adquirir más material por su cuenta.

Algunos links para visitar:

<http://community.borland.com/museum/>

<http://dmoz.org/Computers/Programming/Languages/Pascal/Compilers/>

<http://www.metrowerks.com/>

<http://www.thefreecountry.com/developercity/pascal.html>

<http://www.freepascal.org/download.html>



[Introducción](#)



[Continuar](#)



[Contenido](#)

APRENDA PASCAL

Historia de Pascal

- [Origins](#)
- [Wirth Invents Pascal](#)
- [UCSD Pascal](#)
- [Pascal Becomes Standard](#)
- [Extensions](#)
- [The World Changes](#)
- [So Why Learn Pascal](#)

Origins

Pascal grew out of ALGOL, a programming language intended for scientific computing. Meeting in Zurich, an international committee designed ALGOL as a platform-independent language. This gave them more more free rein in the features they could put into it, but also made it more difficult to write compilers for it. Many computer vendors did not. The lack of compilers on many platforms, combined with its lack of pointers and many basic data types such as characters, led to ALGOL not being widely accepted. Scientists and engineers flocked to FORTRAN, a programming language which *was* available on many platforms. ALGOL mostly faded away except as a language for describing algorithms.

[Back to top](#)

Wirth Invents Pascal

In the 1960s, several computer scientists worked on extending ALGOL. One of these was Dr. Niklaus Wirth of the Swiss Federal Institute of Technology (ETH-Zurich), a member of the original group that created ALGOL. In 1971, he published his specification for a highly-structured language which resembled ALGOL in many ways. He named it *Pascal* after the 17th-century French philosopher and mathematician who built a working mechanical digital computer.

Pascal is very data-oriented, giving the programmer the ability to define custom data types. With this freedom comes strict type-checking, which ensured that data types didn't get mixed up. Pascal was intended as a teaching language, and was widely adopted as such. Pascal is free-flowing, unlike FORTRAN, so student programmers didn't have to worry about formatting. In addition, Pascal reads very much like a natural language, making it very easy to understand code written in it.

[Back to top](#)

UCSD Pascal

One of the things that killed ALGOL was the difficulty of creating a compiler for it. Dr. Wirth avoided this by having his Pascal compiler compile to an intermediate, platform-independent object code stage. Another program turned this intermediate code into executable code. Prof. Ken Bowles at the University of California at San Diego (UCSD) seized on the opportunity this offered to adapt the Pascal compiler to the Apple II, the most popular microcomputer of the day. UCSD P-System became a standard, and was widely used at universities. This was aided by the low cost of Apple II's compared to mainframes, which were necessary at the time to run other languages such as FORTRAN.

[Back to top](#)

Pascal Becomes Standard

By the early 1980's, Pascal has already become widely accepted at universities. Two things happened to make it even more popular.

First, the Educational Testing Service, the company which writes and administers the principal college entrance exam in the United States, decided to add a Computer Science exam to its Advanced Placement exams for high school students. For this exam, it chose the Pascal language. Because of this, secondary-school students as well as college students began to learn Pascal. Pascal remained the official language of the AP exams until 1999, when it was replaced by C++, which gave way to Java very soon afterwards.

Second, a small company named Borland International came out with the Turbo Pascal compiler for the IBM Personal Computer. This compiler was truly revolutionary. It did take some shortcuts and made some modifications to standard Pascal, but these were minor and led to its greatest advantage: speed. Turbo Pascal compiled at a dizzying rate: several thousand lines a minute. At the time, the available compilers for the PC platform were slow and bloated. When Turbo Pascal came out, it was a breath of fresh air. Soon, Turbo Pascal became the *de facto* standard for programming on the PC. When computing magazines published source code for utility programs, it was usually in either assembly or Turbo Pascal.

At the same time, Apple came out with its Macintosh series of computers. Since UCSD Pascal has first been implemented on the Apple II, Apple made Pascal the standard programming language for the Mac. When programmers received the API and example code for Mac programming, it was all in Pascal.

[Back to top](#)

APRENDA PASCAL

Extensions

From version 1.0 to 7.0 of Turbo Pascal, Borland continued to expand the language. One of the criticisms of the original version of Pascal was its lack of separate compilation for modules. Dr. Wirth even created a new programming language, Modula-2, to address that problem. Borland added this to Pascal with its units feature. By version 7.0, many advanced features had been added. One of these was DPMI (DOS Protected Mode Interface), a way to run DOS programs in protected mode, gaining extra speed and breaking free of the 640K barrier instituted by Microsoft its early versions of DOS. Turbo Vision, a text-based windowing system, allowed programmers to create great-looking interfaces in practically no time at all. Pascal even became object-oriented, as version 5.5 adopted the Apple Object Pascal extensions. When Windows 3.0 came out, Borland created Turbo Pascal for Windows, bringing the speed and ease of Pascal to the graphical user interface. It seemed that Pascal's future was secure.

[Back to top](#)

The World Changes

However, this was not so. In the 1970s, Dennis Ritchie and Brian Kernighan of AT&T Bell Laboratories created the C Programming Language. Ritchie then collaborated with Ken Thompson to design the UNIX operating system. AT&T had, at that time, a monopoly on telephone service in the United States, and was permitted to keep that monopoly in exchange for being banned from the computer business. AT&T thus gave away the operating system, with source code, to universities for free.

Thus, a whole generation of computer science students learned Pascal in the introductory programming courses, then learned C when they delved into operating systems. Slowly but surely, C began to filter into the computer programming world.

The killer, ironically enough, was object orientation and the move to Windows on the PC platform. Bjarne Stroustrup introduced object-orientation to most of the world when he created C++. Object Pascal was quickly created in response, but for most programmers, the first thing that pops to mind when OOP is mentioned is C++. At the same time,

As the world moved towards Windows, the job of keeping up with it became greater. Windows was programmed in C (K&R C, to be specific), and Microsoft released the API entirely in C. Example code came in C. Most third-party programs were written in C. Programming Windows applications in Pascal went against the tide.

Many colleges and universities moved away from Pascal, choosing C++, or the new Java, for their programming courses. Finally, the AP exam moved to C++, ending Pascal's dominance in high schools.

[Back to top](#)

So Why Learn Pascal?

Despite its fading away as the *de facto* standard, Pascal is still extremely useful. C and C++ are very symbolic languages. Where Pascal chooses words (e.g. **begin-end**), C/C++ chooses symbols ({-}). Also, C and C++ are not strongly-typed languages. In Pascal, mixing types often led to an error. In C/C++, nothing would happen. You could even treat pointers as integers and do pointer arithmetic with them. In this way, you could very easily crash your program. When the AP exam switched to C++, only a subset of C++ was adopted. Many features, like arrays, were considered too dangerous, and ETS provided its own "safe" version of these features. Java corrects many of these problems of C++ (there are no actual pointers in Java).

Another reason: speed and size. The Borland Pascal compiler is still lightning- fast. Borland has revitalized Pascal for Windows with Delphi, a Rapid-Application-Development environment. Instead of spending several hours writing a user interface for a Windows program in C/C++, you could do it in ten minutes with Delphi's graphical design tools. You could do the same in Visual BASIC, but Delphi is so much faster than Visual BASIC.

Also, Pascal remains preferred at many universities, partly due to the complexity of teaching introductory programming in Java or C++. Teaching Java requires the teaching of handles and object orientation - a lot of overhead for a beginning programming course. To teach simple procedural programming, Pascal remains the top choice.

Thus, even after C, C++, and Java took over the programming world, Pascal retains a niche in the market. Many small-scale freeware, shareware, and open-source programs are written in Pascal/Delphi. So enjoy learning it while it lasts. It's a great introduction to computer programming. It's not scary like C, dangerous like C++, or abstract like Java. In another twenty years, you'll be one of the few computer programmers to know and appreciate Pascal.

[Back to top](#)



[Lección anterior](#)



[Continuar](#)



[Contenido](#)

APRENDA PASCAL

Hola, mundo.

En la corta historia de programación de computadoras, se ha creado una tradición. El primer programa escrito en un nuevo lenguaje siempre es el programa “Hola, mundo”. Copie y pegue el siguiente programa en su editor de texto, luego compílelo y ejecútelo:

```
program Hola;  
begin      (* Main *)  
    writeln ('hola, mundo.')  
end.      (* Main *)
```

La salida debería verse como:

Hola, mundo.



[Lección anterior](#)



[Continuar](#)



[Contenido](#)

APRENDA PASCAL

1A – Estructura de un Programa

La estructura básica de un programa en Pascal es:

```
PROGRAM PnombredelPrograma (FileList);

CONST
    (* Declaración de Constantes *)

TYPE
    (* Declaración de Tipos *)

VAR
    (* Declaración de Variables *)

    (* Definiciones de Subprogramas *)

BEGIN
    (* Instrucciones Ejecutables *)
END.
```

Los elementos de un programa deben estar en el orden correcto, aunque algunos pueden omitirse si no se necesitan. Aquí hay un programa que no hace nada, pero tiene todos los elementos requeridos:

```
program NoHaceNada;
begin
end.
```

Los comentarios en Pascal comienzan con un **(*)** y terminan con un ***)**. No se pueden anidar comentarios :

```
(* (* *) *)
```

dará un error porque el compilador aparea el primer **(*)** con el primer ***)**, ignorando todo lo del medio. El segundo ***)** quedará sin su correspondiente **(*)**.

En Turbo Pascal, **{Comentario}** es una alternativa a **(*) Comentario *)**. La llave abierta significa comienzo de un bloque de comentarios, y la llave cerrada significa el final del mismo.

Los comentarios tienen dos propósitos: primero, hace el código más fácil de entender. Si usted escribe el código sin comentarios, puede volver a él un año después y tendrá mucha dificultad tratando de entender que quiso hacer o porque lo hizo de ese modo. Otro uso de los comentarios es para encontrar errores. Cuando usted no sabe qué está causando un error en su código, puede poner como comentario el segmento del código bajo sospecha de error. Recuerde la restricción anterior respecto de comentarios anidados. No incurrirá en error si hace esto:

```
{ (* Comentario *) }
```

Todos los espacios y los end-of-lines son ignorados por el compilador de Pascal a menos que estén dentro de un string. Sin embargo, para hacer más legible un programa, debería indentar las sentencias y poner sentencias diferentes en distintas líneas.



[Lección anterior](#)



[Continuar](#)



[Contenido](#)

APRENDA PASCAL

1B - Identificadores

Identificadores son nombres que ayudan a referenciar valores almacenados, tales como variables y constantes. Todo programa debe ser **identificado** (de aquí su nombre) por un identificador.

Reglas para identificadores:

- Deben comenzar con una letra del alfabeto inglés.
- Pueden seguir caracteres alfanuméricos y el carácter `_`
- No deben tener caracteres especiales. Ejemplos de caracteres especiales:
`~ ! @ # $ % ^ & * () _ + ` - = { } [] : " ; ' < > ? , . / | \`

Las implementaciones de Pascal pueden diferir en sus reglas sobre caracteres especiales, especialmente el carácter `_`

Varios identificadores son palabras reservadas en Pascal – No puede usarlos como identificadores. Estos son:

and	array	begin	case	const	div	do	downto
else	end	file	for	forward	function	goto	if
in	label	mod	nil	not	of	or	packed
procedure	program	record	repeat	set	then	to	type
until	var	while	with				

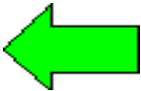
Pascal tiene varios identificadores predefinidos. Pueden reemplazarse con definiciones propias de cada programador, pero entonces desaparecen parte de las funcionalidades de Pascal.

abs	arctan	boolean	char	cos	dispose	Eof	eoln
exp	false	input	integer	ln	maxint	new	odd
ord	output	pack	page	pred	read	readln	real
reset	rewrite	round	sin	sqr	sqrt	succ	text
true	trunc	write	writeln				

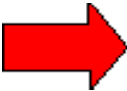
Pascal no distingue entre mayúsculas o minúsculas! `MyProgram`, `MYPROGRAM`, and `mYpRoGrAm` son equivalentes. Pero para hacer los programas más legibles, es buena idea usar mayúsculas. Debido a que C, C++, y Java, los lenguajes de programación dominantes de hoy, distinguen entre mayúsculas y minúsculas, las implementaciones futuras de Pascal pueden venir del mismo modo.

Los identificadores pueden tener cualquier longitud, pero algunos compiladores de Pascal sólo ven cierta longitud de caracteres comenzando desde los primeros. Supongamos que ese número es 32. Entonces, `ABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJ` `ABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJ` `ABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJ` serán equivalentes.

Normalmente uno no utiliza demasiados caracteres o caracteres especiales ya que esto no brinda muchos beneficios a la funcionalidad del programa. Además, permaneciendo dentro de los límites dados por las reglas de selección de identificadores, ayudará en la compatibilidad entre plataformas. La mayoría de los programadores se arreglan con caracteres alfanuméricos.



[Lección anterior](#)



[Continuar](#)



[Contenido](#)

APRENDA PASCAL

1C - Constantes

Las constantes son referenciadas por los identificadores, y pueden tener asignado un valor al comienzo del programa. El valor almacenado en una constante no puede ser modificado.

Las constantes se definen en la sección de constantes:

```
const
  Identificador1 = valor;
  Identificador2 = valor;
  Identificador3 = valor;
```

Ejemplo:

```
const
  Name = 'Tao Yue';
  FirstLetter = 'a';
  Year = 1997;
  pi = 3.1415926535897932;
  UsingNetscapeNavigator = TRUE;
```

El ejemplo muestra los principales tipos de datos permitidos en Pascal: cadenas, caracteres, enteros, reales, y Booleaos. Esos tipos de datos serán explicados en la siguiente sección.

Note que en Pascal los caracteres están encerrados en apostrofes (')!

Las constantes son útiles cuando se sabe que un dato va a cambiar en el futuro. Es más fácil cambiar al principio del programa el dato que contiene esa constante que cambiarlo todas las veces que aparecería a lo largo del programa si no se usara esa constante.

Ejemplo:

```
const PrimerMinistro = 'Winston Churchill';
```

luego cambiará a o:

```
const PrimerMinistro = 'Tony Blair';
```

De este modo el resto del código permanece inalterable, porque se refieren a los datos mediante las constantes. Sólo el valor de las constantes necesita ser modificado.

Las definiciones de constantes también pueden llevar el tipo de datos al que pertenecen. Por ejemplo:

```
const
  a : real = 12;
```

da un identificador a que contiene el valor real 12.0 en lugar del valor entero 12.

Más acerca de tipos de datos en la siguiente sección.



[Lección anterior](#)



[Continuar](#)



[Contenido](#)

APRENDA PASCAL

1D - Variables y Tipo de Datos

Las variables son similares a las constantes, pero sus valores pueden ser cambiados mientras el programa se ejecuta. En Pascal las variables deben declararse antes de que puedan ser usadas:

```
var
  IdentifierList1 : DataType1;
  IdentifierList2 : DataType2;
  IdentifierList3 : DataType3;
  ...
```

IdentifierList es una serie de identificadores, separados por comas (,). Todos los identificadores de la lista son declarados como pertenecientes al mismo tipo de datos.

Los tipos de datos principales en Pascal son:

```
integer
real
char
Boolean
```

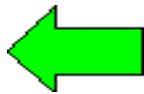
Note que aunque hay constantes que puedan ser cadenas de caracteres (strings), no hay un tipo de datos llamado **string** excepto en algunas implementaciones especializadas (como Turbo Pascal).

Más información sobre tipo de datos en Pascal:

- **integer** puede contener enteros desde -32768 hasta 32767.
- **real** tiene un rango positivo desde 3.4×10^{-38} hasta 3.4×10^{38} . Los valores reales pueden escribirse con notación científica o con notación de punto fijo. Entonces, 452.13 es lo mismo que 4.5213e2
- **char** es para caracteres. Asegurese de encerrarlos caracteres entre apostrofes: 'a' 'B' '+'
Este tipo de datos puede contener caracteres del sistema, como el carácter nulo (0) y el carácter falso-espacio (255).
- **Boolean** admite sólo dos valores:
TRUE y **FALSE**

Un ejemplo de declaración de varias variables es:

```
var
  edad, año, grado : integer;
  circunferencia : real;
  LetterGrade : char;
  DidYouFail : Boolean;
```



[Lección anterior](#)



[Continuar](#)



[Contenido](#)

APRENDA PASCAL

1E – Asignaciones y Operaciones

Una vez declarada una variable, puede almacenar un valor en ella. Esto se llama *asignación*.

Para asignar un valor a una variable, siga ésta sintaxis:

```
nombre_variable := expresión;
```

Note que contrariamente a otros lenguajes, cuyo operador de asignación es el signo igual, Pascal usa dos puntos seguido del signo igual.

La expresión puede ser sólo un valor :

```
un_real := 385.385837;
```

o puede ser una expresión aritmética :

```
un_real := 37573.5 * 37593 + 385.8 / 367.1;
```

Los operadores aritméticos en Pascal son:

Operador	Operación	Operandos	Resultado
+	Suma	real o entero	real o entero
-	Resta	real o entero	real o entero
*	Multipliación	real o entero	real o entero
/	División Real	real o entero	real
div	Division Entero	entero	entero
mod	Módulo	entero	entero

div y **mod** sólo opera con enteros. / trabaja con reales y enteros pero siempre da un resultado real. Las demás operaciones operan con reales y enteros.

Para las operaciones que trabajan con reales y enteros, el resultado será entero si lo son todos los operandos y será real si lo es por lo menos un operando.

Por lo tanto,

```
3857 + 68348 * 38 div 56834
```

dará un entero, pero

```
38573 div 34739 mod 372 + 35730 - 38834 + 1.1
```

será real porque 1.1 es real.

A cada variable sólo se le puede asignar un valor que represente un dato del mismo tipo que el de la variable.

No se le puede asignar un valor real a una variable entera. Sin embargo, hay ciertos tipos de datos que son compatibles con otros. Se le puede asignar un valor entero a una variable real.

Suponga que tiene la siguiente sección de declaración de variables:

```
var
    un_entero : integer;
    un_real : real;
```

Cuando se ejecuta el siguiente bloque de instrucciones,

```
un_entero := 375;
un_real := un_entero;
```

`un_real` tendrá un valor de 375.0, o 3.75e2.

En Pascal el signo menos puede ser utilizado para obtener un valor negativo. El signo mas puede usarse para obtener un valor positivo, sin embargo no se lo hace porque no es necesario dado que por defecto todos los valores positivos.

No deben usarse dos operandos seguidos (uno al lado del otro)!

```
un_real := 37.5 * -2;
```

Esto puede tener sentido para usted ya que está tratando de multiplicar por el número negativo -2. Sin embargo, para Pascal es confuso y no sabrá si multiplicar o restar. Esto se puede evitar usando paréntesis :

```
un_real := 37.5 * (-2);
```

La computadora sigue las operaciones con un orden similar al que usted sigue cuando hace operaciones aritméticas:

* / div mod

+ -

La computadora resuelve cada expresión de acuerdo con las siguientes reglas:

1. Evalua todas las expresiones dentro de los parentesis, comenzando desde el par de parentesis mas interno hacia los de más afuera.
2. Evalua todas las multiplicaciones y divisiones de izquierda a derecha.
3. Evalua todas las sumas y restas de izquierda a derecha.

El valor de

```
3.5 * (2 + 3)
```

seá 17.5.

Pascal no puede realizar operaciones aritméticas standard con datos Booleanos. Hay un conjunto especial de [Operaciones Booleanas](#). Tampoco se deben hacer operaciones standard con datos tipo caracteres porque el resultado puede variar de compilador en compilador.



[Lección anterior](#)



[Continuar](#)



[Contenido](#)

APRENDA PASCAL

1F – Funciones Standard

Pascal tiene varias funciones matemáticas standard que usted puede utilizar. Por ejemplo, para encontrar el valor de seno de π radianes,

```
valor := sin (3.1415926535897932);
```

Note que la función `sin` opera en ángulos expresados en radianes. Esto es valido con todas las funciones trigonométricas. Si todo va bien, el resultado debería dar 0.

Las funciones se llaman mediante el uso del nombre de la función seguido por el argumento(s) entre paréntesis.

Las funciones standard en Pascal son:

Función	Descripción	Tipo de argumento	Tipo de resultado
abs	Valor absoluto	real o entero	Igual al tipo de argumento
arctan	Arcotangente en radianes	real o entero	real
cos	Coseno de un radian	real o entero	real
exp	e a una dada potencia	real o entero	real
ln	Logaritmo natural	real o entero	real
round	Redondeo al entero más próximo	real	entero
sin	Seno de un radian	real o entero	real
sqr	Cuadrado (potencia 2)	real o entero	Igual al tipo de argumento
sqrt	Raiz cuadrada (potencia 1/2)	real o entero	real
trunc	Truncado (redondeo hacia abajo)	real o entero	entero

Para datos de tipo ordinal (entero o carácter), que tiene un claro predecesor y sucesor, pueden utilizarse las siguientes funciones:

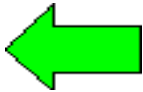
Función	Descripción	Tipo de argumento	Tipo de resultado
chr	caracter con un dado valor ASCII	entero	caracter
ord	Valor ordinal	entero o caracter	entero
pred	Predecesor	entero o caracter	Igual al tipo de argumento
succ	Sucesor	entero o caracter	Igual al tipo de argumento

Un real no es un tipo de dato odinal! Esto es porque no tiene un claro predecesor o sucesor. ¿Cuál es el sucesor de 56.0? Podría ser 56.1 56.01 56.001 56.0001 56.00001 56.000001

Sin embargo, para el entero 56, hay un predecesor definido -- 55 -- y un sucesor definido -- 57.

Lo mismo es válido para caracteres:

```
'b'
Sucesor:  'c'
Predecesor: 'a'
```



[Lección anterior](#)



[Continuar](#)



[Contenido](#)

APRENDA PASCAL

1G - Puntuación e Indentación

Dado que Pascal ignora los espacios y los end-of-lines, hace falta puntuación para decirle al compilador cuando una sentencia termina. Debe haber un punto y coma siguiendo:

1. el encabezamiento del programa
2. cada definición de constantes
3. cada declaración de variable
4. cada definición de tipo
5. todas las sentencias

La última sentencia de un programa, la que precede inmediatamente el END, no requiere punto y coma.

Sin embargo, no hay inconvenientes en poner uno, y evita tener que agregar uno si de pronto se mueve hacia arriba esa sentencia.

Indentación no se requiere. Sin embargo, es una buena práctica para el programador, ya que ayuda a hacer el programa más claro. Se puede tener un programa como el que sigue:

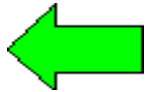
```
program Stupid; const a=5; b=385.3; var alpha,beta:real; begin alpha :=  
a + b; beta:= b / a end.
```

Pero es mucho mejor el siguiente:

```
program Stupid;  
  
    const  
        a = 5;  
        b = 385.3;  
  
    var  
        alpha,  
        beta : real;  
  
    begin      (* main *)  
        alpha := a + b;  
        beta := b / a  
    end.      (* main *)
```

En general, conviene indentar dos o cuatro espacios en cada bloque y dejar una línea en blanco entre bloques (como ente los bloques de constantes const y variables var).

Lo más importante, use comentarios libremente! Si alguna vez usted vuelve a un programa que escribió hace diez años, probablemente no recuerde la lógica con el que lo escribió a menos que lo documente. Es una buena idea comentar la parte ejecutable del programa (main), para distinguirla de los subprogramas.



[Lección anterior](#)



[Continuar](#)



[Contenido](#)

APRENDA PASCAL

1H – Ejemplo N°1

Ahora usted ya sabe usar variables y cambiarles su valor. ¿Listo para su primer programa?

Pero hay un pequeño problema: usted aún no aprendió como mostrar los datos en la pantalla! ¿Cómo va a saber si el programa funciona o no si no puede ver ninguna salida?

Entonces, aquí hay un MUY breve resumen de la próxima lección. Para ver datos en la pantalla, use:

writeln (*argumentos*);

Argumento está compuesto por cualquier cadena de caracteres o nombres de variables separados por comas.

Un ejemplo es:

```
writeln ('La suma da = ', sum);
```

Aquí sigue el programa de ejemplo para el capítulo 1:

Encuentre la suma y el promedio de cinco enteros. El promedio debe ser real. Los cinco números son:

```
45 7 68 2 34
```

Use una constante para representar el número de enteros que manejará el programa – defina alguna constante con el valor 5.

La salida debería ser algo así:

```
Numero de enteros = 5
```

```
Numero1 = 45
```

```
Numero2 = 7
```

```
Numero3 = 68
```

```
Numero4 = 2
```

```
Numero5 = 34
```

```
Suma = 156
```

```
Promedio = 3.1200000000E+01
```

Como puede ver, la salida por defecto de los números reales es la notación científica. En la siguiente sección se verá como formatearlo para fijar el punto decimal.

Para ver una posible solución, haga click en Continuar.



[Lección anterior](#)



[Continuar](#)



[Contenido](#)

APRENDA PASCAL

1Ha - Solución

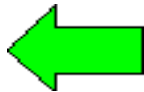
```
(* Author:      Tao Yue
   Date:        19 June 1997
   Description:
       Find the sum and average of five predefined numbers
   Version:
       1.0 - original version
*)

program SumAverage;

const
    NumberOfIntegers = 5;

var
    A, B, C, D, E : integer;
    Sum : integer;
    Average : real;

begin    (* Main *)
    A := 45;
    B := 7;
    C := 68;
    D := 2;
    E := 34;
    Sum := A + B + C + D + E;
    Average := Sum / 5;
    writeln ('Number of integers = ', NumberOfIntegers);
    writeln ('Number1 = ', A);
    writeln ('Number2 = ', B);
    writeln ('Number3 = ', C);
    writeln ('Number4 = ', D);
    writeln ('Number5 = ', E);
    writeln ('Sum = ', Sum);
    writeln ('Average = ', Average)
end.    (* Main *)
```



[Lección anterior](#)



[Continuar](#)



[Contenido](#)

APRENDA PASCAL

2A - Entrada

Entrada significa leer un dato desde la memoria, del teclado, del mouse o de un archivo en un disco. No veremos cómo leer un dato proveniente del mouse porque la sintaxis difiere de máquina a máquina, y puede requerir una interrupción complicada al driver del mouse. Si desea ver el código fuente para un mouse para DOS, vea el siguiente link:

<http://web.mit.edu/taoyue/www/tutorials/pascal/pas-mou.html>

Esta unidad no trabajará bajo Windows, Macintosh, o X-Windows, porque estos sistemas operativos manejan toda entrada del mouse por usted y no le permiten una interface con el mouse directamente.

El formato básico para leer entradas de datos es:

read (*Lista_de_Variables*);

Variable_List es una serie de identificadores de variables separados por comas.

read, sin embargo, no va a la siguiente línea. Esto puede ser un problema con entradas de caracteres, porque el carácter de end-of-line es leído como un espacio.

Para leer datos de entrada y luego ir a la siguiente línea:

readln (*Lista_de_Variables*);

Suponga que lee los enteros *a*, *b*, *c*, y *d*.

45 97 3

1 2 3

Esto será el resultado de varias sentencias:

Sentencia(s)	a	b	c	d
read (a); read (b);	45	97		
readln (a); read (b);	45	1		
read (a, b, c, d);	45	97	3	1
readln (a, b); readln (c, d);	45	97	1	2

La sentencia **read** no salta a la siguiente línea, mientras que **readln** siempre salta a la siguiente línea luego de cada lectura. Cuando se leen enteros, se saltan todos los espacios hasta que se encuentra un número. A partir de ahí, todo número subsecuente es leído, hasta que se alcance un carácter no numérico (incluido el espacio).

8352.38

Cuando se lee un entero, el valor leído es 8352.

Inmediatamente después lee el carácter ' . '.

Si trata de leer ese número en dos enteros, tampoco funcionará porque cuando la computadora busca el segundo dato, ve ' . ' y para avisando que no puede encontrar ningún dato para leer.

Con valores reales, la computadora también salta los espacios en blanco y luego lee todo lo que pueda ser leído. Sin embargo, hay una restricción: un real sin parte entera debe comenzar con 0.

.678

es inválido y la computadora no lo puede leer como real, pero

0.678

es correcto.

Asegúrese que todos los identificadores en la lista de argumentos se refieran a variables! En el argumento de una instrucción de lectura de datos de entrada no puede haber constantes ni valores literales.



[Lección anterior](#)



[Continuar](#)



[Contenido](#)

APRENDA PASCAL

2B - Salida

Para escribir datos en la pantalla, también hay dos sentencias:

write (*Lista_de_Argumentos*);

writeln (*Lista_de_Argumentos*);

La sentencia `writeln` salta a la línea siguiente cuando termina.

En la lista de argumentos se pueden usar cadenas de caracteres, constantes o valores literales. Para que aparezcan comillas con una cadena de caractres, se deben usar dos comillas consecutivas: ``



[Lección anterior](#)



[Continuar](#)



[Contenido](#)

APRENDA PASCAL

2C – Formato de Salida

Para cada identificador o valor literal en la lista de argumentos use:

Valor : cantidad_de_digitos_o_ancho_de_campo

La salida quedará justificada en una cantidad de enteros igual a los especificados. Si la cantidad de dígitos especificada no es suficiente para el dato que se va a escribir, dicha especificación es ignorada y el dato se muestra en toda su longitud (excepto para valores reales—vea a continuación).

Suponga que tenemos:

```
write ('Hi':10, 5:4, 5673:2);
```

La salida (con un guiones simulando los espacios) será:

```
-----Hi---55673
```

Para valores reales, se puede usar la sintaxis previamente mencionada para mostrar notación científica en un ancho de campo especificado., o puede convertirse a notación fija con:

Valor : ancho_de_campo_o_cantidad_de_digitos : ancho_de_campo_decimal

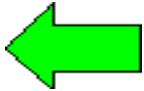
El ancho de campo es el ancho total, incluida la parte decimal. La parte entera del número siempre se muestra con todos los dígitos, incluso si no tiene suficiente espacio asignado. Sin embargo, si el número de dígitos decimales excede el espacio especificado, la salida se redondea al número de dígitos especificado en el ancho de campo decimal (aunque la variable en sí misma no cambia).

Entonces

```
write (573549.56792:20:2);
```

se verá como:

```
-----573549.57
```



[Lección anterior](#)



[Continuar](#)



[Contenido](#)

APRENDA PASCAL

2D - Archivos

El uso de archivos para entrada/salida es diferente según el compilador de Pascal que se emplee. Sin embargo, el modo de lectura/escritura en archivos es el mismo en todos los compiladores:

```
read (file_variable, argument_list);  
write (file_variable, argument_list);
```

Lo mismo con readln y writeln.

La mayoría de los sistemas Pascal requieren que se declare un archivo en la sección de variables:

```
var  
...  
filein, fileout : text;
```

Usualmente se usa el tipo de datos texto. La diferencia entre compiladores de Pascal aparece luego de abrir un archivo para lectura o escritura. En la mayoría de los compiladores de Pascal, incluido Metrowerks Codewarrior y el traductor de Pascal incluido con Unix/Linux, usan:

```
reset (file_variable, 'filename.extension');
```

para abrir un archivo para lectura.

Usan:

```
rewrite (file_variable, 'filename.extension');
```

para abrir un archivo para escritura. Note que no se puede abrir un archivo para ambas cosas (lectura y escritura). Un archivo abierto con *reset* sólo puede ser usado con *read* y *readln*. Un archivo abierto con *rewrite* sólo puede ser usado con *write* y *writeln*.

Turbo Pascal hace esto de modo diferente. Primero se debe asignar un archivo a una variable, luego se llama a *reset* o *rewrite* usando sólo la variable.

```
assign (file_variable, 'filename.extension');  
reset (file_variable)
```

El modo de representar el path difiere dependiendo del sistema operativo. Ejemplos:

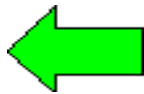
```
Unix:    ~login_name/filename.ext  
DOS/Win: c:\directory\name.pas  
Mac:     Disk_Name:Programs Directory:File Name.pas
```

Antes de terminar el programa se deben cerrar los archivos! Esto es igual para todos los compiladores:

```
close (File_Identifier);
```

El siguiente programa usa archivos. Fue escrito para Turbo Pascal y DOS, y crea file2.txt con el primer carácter proveniente de file1.txt:

```
program CopyOneByteFile;  
  
var  
    mychar : char;  
    filein, fileout : text;  
  
begin  
    assign (filein, 'c:\file1.txt');  
    reset (filein);  
    assign (fileout, 'c:\file2.txt');  
    rewrite (fileout);  
    read (filein, mychar);  
    write (fileout, mychar);  
    close(filein);  
    close(fileout)  
end.
```



[Lección anterior](#)



[Continuar](#)



[Contenido](#)

APRENDA PASCAL

2E - EOLN y EOF

EOLN es una función Booleana que es verdadera (TRUE) cuando se alcanza el final de una línea (end of a line) en un archivo de entrada abierto.

eoln (*file_variable*)

Para ver si la entrada standard (el teclado) está en un end-of-line, simplemente poner **eoln** sin ningún parametro:

eoln

EOFTTRUE cuando se alcanza el final del archivo.

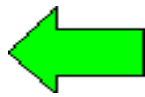
eof (*file_variable*)

o

eof

para la entrada standard.

Normalmente no se pulsa el carácter end-of-file desde el teclado. En máquinas DOS/Windows, este carácter es Control-Z or F6. En máquinas UNIX/Linux, el carácter end-of-file es Control-D.



[Lección anterior](#)



[Continuar](#)



[Contenido](#)

APRENDA PASCAL

2F – Ejemplo N°2

Nuevamente encuentre la suma y el promedio de cinco números, pero esta vez lea los cinco enteros y muestre la salida ordenada en columnas.

Vea el [problema original](#) si lo necesita. Los números son:

45 7 68 2 34

La salida ahora debería verse del siguiente modo:

Numero de enteros = 5

Numero1: 45

Numero2: 7

Numero3: 68

Numero4: 2

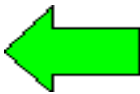
Numero5: 34

=====

Suma: 156

Promedio: 31.2

Como un ejercicio extra, pruebe escribir la salida del programa en un archivo.



[Lección anterior](#)



[Continuar](#)



[Contenido](#)

APRENDA PASCAL

2Fa - Solución

```
(* Author:      Tao Yue
   Date:        19 June 1997
   Description:
       Find the sum and average of five predefined numbers
   Version:
       1.0 - original version
       2.0 - read in data from keyboard
*)

program SumAverage;

const
    NumberOfIntegers = 5;

var
    A, B, C, D, E : integer;
    Sum : integer;
    Average : real;
    fileout : text;

begin    (* Main *)
    write ('Enter the first number: ');
    readln (A);
    write ('Enter the second number: ');
    readln (B);
    write ('Enter the third number: ');
    readln (C);
    write ('Enter the fourth number: ');
    readln (D);
    write ('Enter the fifth number: ');
    readln (E);
    Sum := A + B + C + D + E;
    Average := Sum / 5;
    writeln ('Number of integers = ', NumberOfIntegers);
    writeln;
    writeln ('Number1:', A:8);
    writeln ('Number2:', B:8);
    writeln ('Number3:', C:8);
    writeln ('Number4:', D:8);
    writeln ('Number5:', E:8);
    writeln ('=====');
    writeln ('Sum:', Sum:12);
    writeln ('Average:', Average:10:1);
    assign (fileout, 'D:\WD\Pascal\Ejercicios\Prog13\promedio.txt');
    rewrite (fileout);
    writeln (fileout, 'Numero1:', A:8);
    writeln (fileout, 'Numero2:', B:8);
    writeln (fileout, 'Numero3:', C:8);
    writeln (fileout, 'Numero4:', D:8);
    writeln (fileout, 'Numero5:', E:8);
    writeln (fileout, '=====');
    writeln (fileout, 'Suma:', Sum:12);
    writeln (fileout, 'Promedio:', Average:10:1);
    close(fileout);
end.    (* Main *)
```



[Lección anterior](#)



[Continuar](#)

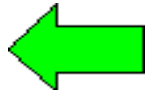


[Contenido](#)

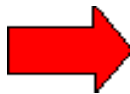
APRENDA PASCAL

3A – Control Secuencial

Control secuencial es lo normal. La computadora ejecuta cada sentencia y sigue hacia la siguiente hasta que ve un end.



[Lección anterior](#)



[Continuar](#)



[Contenido](#)

APRENDA PASCAL

3B - Expresiones Booleanas

Las expresiones Booleanas se utilizan para comparar dos valores.

La forma más simple de una expresión Booleana es:

```
valor1 operador_relacional valor2
```

The following relational operators are used:

<	Menor que
>	Mayor que
=	Igual a
<=	Menor o igual que
>=	Mayor o igual que
<>	No igual a

Se pueden asignar expresiones Booleanas a variables Booleanas:

```
some_bool := 3 < 5;
```

Por supuesto,el valor de some_bool es VERDADERO (TRUE).

Expresiones Booleanas complejas se forman utilizando los operadores Booleanos:

not	Negación (~)
and	Conjunción (^)
or	OR (v)
xor	OR-exclusivo

NOT es un operador unitario – se lo aplica a un único valor y lo invierte:

```
not verdadero = falso
```

```
not falso = verdadero
```

AND da VERDADERO sólo si ambas expresiones son VERDADERAS.

```
VERDADER and FALSO = FALSO                      VEDADERO and VERDADERO = VERDADERO
```

OR da VERDADERO si al menos una expresió es VEDADERA, o si lo son ambas. Lo siguiente es

VERDADERO:

```
VERDADERO or VERDADERO
```

```
VERDADERO or FALSO
```

```
FALSO or VERDADERO
```

XOR da VERDADERO si una de las expresiones es VERDADERA y la otra es FALSA. Por lo tanto,

```
VERDADERO xor VERDADERO = FALSO
```

```
VERDADERO xor FALSO = VERDADERO
```

```
FALSO xor VERDADERO = VERDADERO
```

```
FALSO xor FALSO = FALSO
```

Cuando combine dos expresiones Booleanas utilizando operadores Booleanos y relacionales, sea cuidadoso en el uso de paréntesis.

```
(3>5) or (650<1)
```

Esto es porque los operadores Booleanos tienen on orden de precedencia mayor que los operadores relacionales:

```
not
```

```
* / div mod and
```

```
+ - or
```

```
< > <= >= = <>
```

De este modo,

```
3 > 5 or 650 < 1
```

es evaluado como

```
3 > (5 or 650) < 1
```

lo que no tiene sentido porque los operadores Booleanos sólo operan con valores Booleanos, no con enteros.

Los operadores Booleanos (AND, OR, NOT, XOR) pueden ser utilizados en variables Booleanas del mismo modo que son utilizados en expresiones Booleanas.

Siempre que sea posible, no compare dos valores reales con el signo igual. Pequeños errores de redondeo pueden provocar que dos expresiones equivalentes difieran.



[Lección anterior](#)



[Continuar](#)



[Contenido](#)

APRENDA PASCAL

3Ca - IF

La sentencia IF permite una bifurcción o selección de que camino seguir en función del resultado de una operación Booleana.

La sintaxis para una selección incompleta o bifurcación de camino simple es:

```
if ExpresiónBooleana then  
SentenciaSiVerdadero;
```

La sentencia se ejecuta si la expresión Booleana se evalúa como verdadera. De otro modo, se la salta.

La sentencia IF acepta sólo una sentencia a ejecutar. Si se desean ejecutar más de una sentencia, debe usarse begin-end de modo que las incluya:

```
if ExpresiónBooleana then  
begin  
Sentencia1;  
Sentencia2  
end;
```

La sintaxis para una selección completa o bifurcación de camino doble es:

```
if ExpresiónBooleana then  
SentenciaSiVerdadero  
else  
SentenciaSiFalso;
```

Si la expresión Booleana se evalúa como FALSA, se ejecutará la sentencia que sigue al else. **NO** debe usarse punto y coma luego de la sentencia que precede al else. Un punto y coma en dicha sentencia probocaría que la computadora interprete al if como selección incompleta o bifurcación de camino simple.

Para selección de caminos múltiples, simplemente deben anidarse sentencias if:

```
if Condición1 then  
    Sentencia1  
else  
    if Condición2 then  
        Sentencia2  
    else  
        Sentencia3;
```

Sea cuidadoso con las anidaciones. A veces la computadora no sabrá que es lo que usted quiere hacer:

```
if Condición1 then  
    if Condición2 then  
        Sentencia2  
else  
    Sentencia1;
```

El else siempre se relaciona al if más reciente, por lo tanto la computadora interpretará el bloque de código precedente como:

```
if Condición1 then  
    if Condición2 then  
        Sentencia2  
    else  
        Sentencia1;
```

Que no es lo que se deseaba hacer. Se puede usar un else con una sentencia nula:

```
if Condición 1 then  
    if Condición 2 then  
        Sentencia2  
    else  
else  
    Statement1;
```

O bien se puede usar un bloque begin-end. Pero la mejor manera de clarificar el código anterior es reescribir la condición del primer if como:

```
if not Condición1 then  
    Sentencia1  
else  
    if Condición2 then  
        Sentencia2;
```

El ejemplo anterior muestra en qué caso el operador not es práctico. Si la Condición1 fuera una expresión Booleana como: (not(a < b) or (c + 3 > 6)) and g invertir esta expresión sería mucho más complicado que negarla mediante un not.



[Lección anterior](#)



[Continuar](#)



[Contenido](#)

APRENDA PASCAL

3Cb - CASE

Suponga que se desea una selección incompleta si **b** es 1, 7, 2037, o 5; y que el programa siga otro camino en caso contrario. Esto podría hacerse así:

```
if (b = 1) or (b = 7) or (b = 2037) or (b = 5) then
  Sentencia1
else
  Sentencia2;
```

Pero debería ser más simple listar los números para los cuales se desea que la Sentencia1 se ejecute. Esto se puede hacer con la sentencia case:

```
case b of
  1,7,2037,5: Sentencia1;
  otherwise  Sentencia2
end;
```

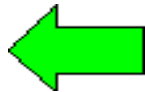
La sintaxis general para la sentencia case es:

```
case selector of
  Lista1: Sentencia1;
  Lista2: Sentencia2;
  ...
  Listan: Sentencia;
otherwise Sentencia
end;
```

La parte otherwise es opcional y difiere de compilador en compilador. En muchos compiladores se utiliza la palabra **else** en lugar de otherwise.

selector es una variable de tipo ordinal. No se pueden usar reales!

La lista debe consistir en valores literales. Puedn usarse constantes pero no variables.



[Lección anterior](#)



[Continuar](#)



[Contenido](#)

APRENDA PASCAL

3Da - FOR..DO

Los bucles nos sirven para repetir unas mismas sentencias varias veces, hasta que se cumpla alguna condición.

Hay tres tipos de bucles:

- **repetición fija** – se repite un número de veces fija
- **pretest** – verifica una expresión Booleana, si es VERDADERA va al bucle
- **posttest** – ejecuta el bucle, luego verifica una expresión Booleana

En Pascal, el bucle de repetición fija es el bucle for-to. Su sintaxis general es:

```
for indice := InicioBajo to FinalAlto do  
sentencia;
```

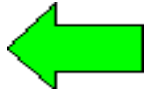
La variable *indice* debe ser un dato de tipo ordinal. Puede usarse el *indice* en calculos dentro del bucle, pero no se le debe cambiar su valor. Un ejemplo deluso de un indice es:

```
suma := 0;  
for contador := 1 to 100 do  
    suma := suma + contador;
```

En el bucle for-to el valor inicial del indice debe ser menor que el valor final o el bucle nunca se ejecutará! Si se desea contar hacia abajo, se debe usar el bucle for-downto:

```
for indice := InicioAlto downto FinalBajo do  
sentencia;
```

En Pascal, el bucle for-to sólo puede contar con incrementos (pasos) de 1.



[Lección anterior](#)



[Continuar](#)



[Contenido](#)

APRENDA PASCAL

3Db - WHILE..DO

El lazo con condición de fin previa (**pretest**) tiene el siguiente formato:

```
while ExpresiónBooleana do  
sentencias;
```

El lazo continúa ejecutandose hasta que la expresión Booleana sea FALSA. Dentro del cuerpo del lazo se puede modificar las variables usadas en la expresión Booleana afectando el resultado de dicha comparación.

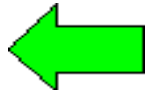
Ejemplo: Un lazo infinito

```
a := 5;  
while a < 6 do  
  writeln (a);
```

Remedie esta situación cambiando el valor de la variable:

```
a := 5;  
while a < 6 do  
  begin  
    writeln (a);  
    a := a + 1  
  end;
```

El lazo WHILE ... DO se llama lazo de pretest porque la condición de finalización se realiza al comienzo de la ejecución del cuerpo del lazo. Por lo tanto si la condición de finalización comienza como FALSA, el cuerpo del lazo while nunca se ejecuta.



[Lección anterior](#)



[Continuar](#)



[Contenido](#)

APRENDA PASCAL

3Dc - REPEAT..UNTIL

El lazo con condición de finalización posterior (**posttest**) tiene el siguiente formato:

```
repeat  
sentencia1;  
sentencia2  
until ExpresiónBooleana;
```

Dentro de un lazo *repeat* se pueden poner sentencias compuestas sin necesidad de usar *begin-end*.

El lazo *repeat* continúa hasta que la expresión Booleana sea VERDADERA, mientras que un lazo *while* continúa hasta que la expresión Booleana sea FALSA.

Este lazo se llama lazo *posttest* porque la condición de finalización se verifica después de que se ejecuta el cuerpo del lazo. El lazo **REPEAT** es útil cuando se desea ejecutar al menos una vez el lazo, al margen de la condición de finalización dada por el valor inicial de la expresión Booleana.



[Lección anterior](#)



[Continuar](#)



[Contenido](#)

APRENDA PASCAL

3E – Ejemplo N°3

Problema 1

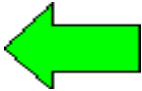
Encuentre los 10 primeros números en la serie de Fibonacci. La serie de Fibonacci comienza con dos números:

1 1
Cada número siguiente se forma sumando los dos números previos.
1+1=2, 1+2=3, 2+3=5, etc. Esto forma la siguiente secuencia:
1 1 2 3 5 8 13 21 34 55

Problema 2

Muestre todas las potencias de 2 que sean <= 20000. Muestre la lista en un formato apropiado, con comas entre los números y 5 números por línea. La salida debe verse del siguiente modo:

1, 2, 4, 8, 16,
32, 64, 128, 256, 512,
1024, 2048, 4096, 8192, 16384



[Lección anterior](#)



[Continuar](#)



[Contenido](#)

APRENDA PASCAL

3Ea - Soluciones

Solución al Problema de la Serie de Fibonacci

```
(* Author:  Tao Yue
   Date:      19 July 1997
   Description:
       Find the first 10 Fibonacci numbers
   Version:
       1.0 - original version
*)

program Fibonacci;

var
    Fibonacci1, Fibonacci2 : integer;
    temp : integer;
    count : integer;

begin    (* Main *)
    writeln ('First ten Fibonacci numbers are:');
    count := 0;
    Fibonacci1 := 0;
    Fibonacci2 := 1;
    repeat
        write (Fibonacci2:7);
        temp := Fibonacci2;
        Fibonacci2 := Fibonacci1 + Fibonacci2;
        Fibonacci1 := Temp;
        count := count + 1
    until count = 10;
    writeln;

    (* Of course, you could use a FOR loop or a WHILE loop
       to solve this problem. *)

end.    (* Main *)
```

Solución al Problema de Potencias de Dos

```
(* Author:  Tao Yue
   Date:      13 July 2000
   Description:
       Display all powers of two up to 20000, five per line
   Version:
       1.0 - original version
*)

program PowersofTwo;

const
    numperline = 5;
    maxnum = 20000;
    base = 2;

var
    number : longint;
    linecount : integer;

begin    (* Main *)
    writeln ('Powers of ', base, ', 1 <= x <= ', maxnum, ':');
    (* Set up for loop *)
    number := 1;
    linecount := 0;
    (* Loop *)
    while number <= maxnum do
        begin
            linecount := linecount + 1;
            (* Print a comma and space unless this is the first
               number on the line *)
            if linecount > 1 then
                write (', ');
            (* Display the number *)
```

APRENDA PASCAL

```
write (number);
(* Print a comma and go to the next line if this is
   the last number on the line UNLESS it is the
   last number of the series *)
if (linecount = numperline) and not (number * 2 > maxnum) then
begin
    writeln (',');
    linecount := 0
end;
(* Increment number *)
number := number * base;
end;  (* while *)
writeln;

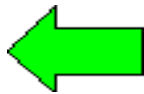
(* This program can also be written using a
   REPEAT..UNTIL loop. *)
```

end. (* Main *)

Hay tres constantes: la base, el número de potencias a mostrar en cada línea y el número máximo. Esto asegura que el programa puede ser adptado facilmente en el futuro.

La utilización de constantes en vez de letras es un buen hábito de programación. Cuando escriba programas realmente largos, puede referirse a ciertos números cientos de veces y si no tienen nombres prácticos, será difícil encontrarlos. También podría ocurrir que use el mismo valor en un contexto diferente, haciendo imposible una búsqueda y reemplazo global. Usando constantes se hace fácil expandir el programa.

Note que la variable number es del tipo longint. Esto es por si number toma el valor 32768, la siguiente potencia de dos luego de 16384. Esto excede el rango de integer type: -32768 to 32767.



[Lección anterior](#)



[Continuar](#)



[Contenido](#)

APRENDA PASCAL

4A - Procedimientos

Un procedimiento es un subprograma. Un subprograma ayuda a reducir la cantidad de redundancia en un programa. A las sentencias que se ejecutan una y otra vez en un programa pero que no estan contenidas en un bucle, frecuentemente se las pone en un subprograma.

Los subprogramas también facilitan el diseño de un programa permitiendo hacerlo desde lo más general hacia lo más específico (top-down design). Por ejemplo, un programa para ir de una habitación a otra sería:

- I. Salir de la primer habitación
- II. Ir a la segunda habitación
- III. Entrar en la segunda habitación

El paso I puede desglosarse en:

- I. Salir de la primer habitación
 - A. Ir hacia la puerta
 - B. Abrir la puerta
 - C. Salir por la puerta
 - D. Cerrar la puerta

...

Ir hacia la puerta puede dividirse en:

- A. Ir hacia la puerta
 - 1. Levantarse del asiento
 - 2. Girar hacia la puerta
 - 3. Caminar hasta la puerta

Esto a su vez puede ser desglosado en más y más pasos. Podrá parecer muy detallado, pero para programación de computadoras esto es, en efecto, lo que se tiene que hacer. La computadora no puede entender sentencias generales -- se debe ser específico.

Las tareas principales deben estar contenidas en procedimientos, de modo que en el cuerpo principal del programa (main), no haya que preocuparse por detalles. Esto también hace al código reusable. Se pueden guardar los procedimientos en un archivo y llamarlos desde el programa.

Un procedimiento tiene la misma forma básica que un programa:

procedure *Nombre*;

```
const
  (* Constantes *)

var
  (* Variables *)

begin
  (* Sentencias *)
end;
```

En el end hay un punto y coma en vez de un punto. Para llamar al procedimiento desde el programa, solo hace falta usar su nombre como `writeln` por ejemplo.

Nombre;

Los procedimientos son usados muy frecuentemente para la salida de datos.



[Lección anterior](#)



[Continuar](#)



[Contenido](#)

APRENDA PASCAL

4B - Parámetros

Se puede incluir una lista de parámetros como parte del encabezamiento de un procedimiento. La lista de parámetros permite transferir valores de variables desde el programa principal hacia el procedimiento.

El encabezamiento del procedimiento con parámetros es:

procedure Nombre (*lista_de_parámetros_formal*);

La lista de parámetros formal consiste de varios grupos de parámetros separados por punto y comas:

grupo_de_param_1; grupo_de_param_2; ... ; grupo_de_param_n

Cada grupo de parámetros tiene la forma:

identificador_1, identificador_2, ... , identificador_n : data_type

Al procedimiento se lo llama pasando los argumentos (lista de parámetros actual) del mismo número y tipo que la lista de parámetros formal.

Un ejemplo es:

```
procedure Nombre (a, b : integer; c, d : real);
begin
  a := 10;
  b := 2;
  writeln (a, b, c, d)
end;
```

Suponga que llama al procedimiento desde el programa:

```
alfa := 30;
Nombre (alfa, 3, 4, 5);
```

¿Cuál es el valor de alfa al retornar al programa luego de haber llamado al procedimiento? 30. El valor de alfa en el programa no se ve afectado por lo que pase dentro del procedimiento. Dentro del procedimiento alfa fue pasado a a que tiene un valor de 10, por lo que en writeln a siempre será 10.

Este modo de pasar parámetros se llama **por valor**. Se pasa el **valor** de la variable al procedimiento.

Otro modo de pasar parámetros es **por-referencia**. Este modo crea un vínculo entre el parámetro formal y el parámetro actual. Cuando en el procedimiento se modifica el parámetro formal, en el programa principal el parámetro actual también se modifica. Para llamar al procedimiento **por-referencia** el grupo de parámetros debe estar precedido por VAR:

VAR *identificador1, identificador2, ..., identificadorn : tipo_de_dato;*

En este caso no se permite el uso de constantes y letras como parámetro actual en el programa principal porque pueden ser modificados en el procedimiento.

Ejemplo de procedure:

```
procedure Nombre(a, b : integer; VAR c, d : integer);
begin
  c := 3;
  a := 5
end.
```

Ejemplo de parte del programa principal:

```
alfa := 1;
gama := 50;
delta := 30;
Nombre (alfa, 2, gama, delta);
```

Luego de que el procedimiento se ejecute, gama tendrá el valor 3 porque c es un parámetro por referencia, pero alfa seguirá siendo 1 porque es un parámetro por valor.

Esto es un poco confuso. Llamar al procedimiento pasando parámetros por valor es como darle al procedimiento una copia del valor. El procedimiento trabaja con esa copia y cuando termina la descarta. La variable original no se modifica.

Llamar al procedimiento por referencia es darle al procedimiento la variable. El procedimiento trabaja directamente con la variable y la devuelve al programa principal.

En otras palabras, llamar al procedimiento por valor es una transferencia de datos de un solo sentido: del programa principal al procedimiento. Llamarlo por referencia implica una transferencia de datos en ambos sentidos: del programa al procedimiento y del procedimiento al programa.



[Lección anterior](#)



[Continuar](#)



[Contenido](#)

APRENDA PASCAL

4C - Funciones

Las funciones operan del mismo modo que los procedimientos, pero siempre retornan al programa principal con un valor obtenido a partir de las variables pasadas a la función y procesadas por esta:

function *Nombre* (*lista_de_parámetros*) : *tipo_de_variable_de_retorno*;

A las funciones se las llama desde el programa usandolas en expresiones:

a := Nombre (5) + 3;

Se debe tener cuidado de no usar el nombre de la función del lado derecho de una ecuación dentro de la función. Esto es:

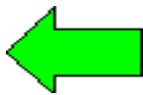
```
function Nombre : integer;  
begin  
  Nombre := 2;  
  Nombre := Nombre + 1  
end.
```

En lugar de dar un 3, como puede esperarse, esto genera un bucle recursivo infinito. Nombre que llama a Nombre, que llama a Nombre, que llama a Nombre, etc.

El valor de retorno de la función se obtiene asignando un valor al identificador de la función.

Nombre := 5;

Generalmente es una mala práctica de programación el utilizar VAR dentro de los parámetros de una función -- las funciones deben retornar solamente un valor. Usted seguramente no desea que la función seno (sin) cambie los pi radianes de su programa a 0 porque son equivalentes--usted solo desea la respuesta 0.



[Lección anterior](#)



[Continuar](#)



[Contenido](#)

APRENDA PASCAL

4D - Scope

Scope es un método para encontrar en forma precisa que variables son accesibles a cada subprograma. Los gráficos scope consisten en cajas dentro de cajas dentro de cajas. Cada caja tiene su nombres y contienen identificadores dentro de las mismas, mostrando que ellos pueden ser accedidos en esa caja y en todas las cajas que estan adentro de esta.

Usted tiene procedimientos dentro de procedimientos, variables dentro de procedimientos, y su trabajo es tratar de entender cuando cada variable puede ser vista por el procedimiento.

Una variable global es una variable definida en el programa principal. Cualquier subprograma puede verla, usarla, y modificarla. Todos los subprogramas pueden llamarse a si mismos, y pueden llamar a otros subprogramas definidos antes que ellos.

El punto principal es: dentro de cualquier bloque de código (procedimiento, función o donde sea), los unicos identificadores que son accesibles son aquellos definidos *antes* que ese bloque, ya sea *interior* o *exterior* al mismo.

Otra cosa:

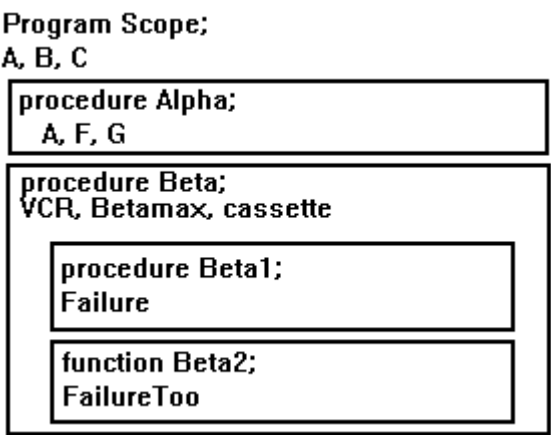
```
program Stupid;
var A;
  procedure StupidToo;
  var A;
  begin
    A := 10;
    writeln (A)
  end;
begin (* Main *)
  A := 20;
  writeln (A);
  StupidToo;
end.  (* Main *)
```

La salida es:

```
20
10
```

Si dos variables con el mismo identificador son declaradas en un subprograma y en el programa principal, cada uno de ellos vera la suya: el programa principal verá la definida en el programa principal y el subprograma la definida en el subprograma (no la definida en el programa principal). La definición usada por un identificador es SIEMPRE la definida como MAS LOCAL.

El siguiente gráfico es un gráfico scope:



- Todos los bloques pueden ver las variables globales A, B, y C.
- En el procedimiento Alpha la definición global de A es reemplazada por una definición local.
- Beta1 y Beta2 pueden ver a las variables VCR, Betamax, y cassette.
- Beta1 no puede ver a las variables FailureToo, y Beta2 no puede ver a Failure.
- Solamente el subprograma Alpha puede acceder a F y G.
- El procedimiento Beta puede llamar a Alpha y a Beta.
- La función Beta2 puede llamar a cualquier subprograma, incluso a si misma (el programa principal no es un subprograma).



[Lección anterior](#)



[Continuar](#)



[Contenido](#)

APRENDA PASCAL

4E - Recursividad

Recursividad es un tema de difícil comprensión. Sin embargo, es muy fácil de aplicar cuando se lo entiende. El ejemplo de la lección 4G se refiere a recursividad.

Recursividad significa permitir a una función o procedimiento llamarse a sí mismo. Estos pueden seguir llamándose a sí mismos hasta alcanzar algún límite.

La función sumatoria, designada por la letra Sigma mayúscula en matemática, es un ejemplo popular de recursividad:

```
function Sumatoria (num : integer) : integer;  
begin  
    if num = 1 then  
        Sumatoria := 1  
    else  
        Sumatoria := Sumatoria (num-1) + num  
    end;  
end;
```

Suponga que llama a la función *Sumatoria* para 3.

a := Sumatoria(3);

- Sumatoria(3) se convertirá en Sumatoria(2) + 3.
- Sumatoria(2) se convertirá en Sumatoria(1) + 2.
- En 1, la recursividad se detendrá.
- Sumatoria(2) será $1 + 2 = 3$.
- Sumatoria(3) será $3 + 3 = 6$.
- a valdrá 6.

La recursividad se produce hacia atrás hasta que se alcanza un punto en el cual la respuesta queda definida en 1, luego va hacia delante con esa definición encontrada, resolviendo las definiciones planteadas hasta llegar a 1.

Muy importante! Todo procedimiento/función recursivo *debe* tener algún tipo de condición que la detenga. Bajo la condición llamada **condición base**, la recursividad debería detenerse. Bajo toda otra condición la recursividad podría ser infinita. En el ejemplo anterior, la condición base fue $\text{num} = 1$. Sin **una condición base**, la recursividad no tendrá lugar o será infinita.



[Lección anterior](#)



[Continuar](#)



[Contenido](#)

APRENDA PASCAL

4F – Forward

¿Recuerda que los procedimientos/funciones sólo pueden ver variables y otros subprogramas que ya hayan sido definidos?

Hay una excepción.

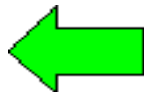
Si hay dos subprogramas, cada uno de los cuales llama al otro, se tiene el dilema de que no importa cual subprograma se ponga primero, el segundo no podrá ser llamado por el primero.

Para resolver este problema del huevo o la gallina, se usa la referencia forward.

```
procedure Tarde (lista de parametros); forward;

procedure Temprano (lista de parametros);
begin
    ...
    Tarde (lista de parametros);
end;
...
procedure Tarde;
begin
    ...
    Temprano (lista de parametros);
end;
```

Lo mismo sirve para funciones. Simplemente hay que agregar **forward**; al final del encabezamiento.



[Lección anterior](#)



[Continuar](#)

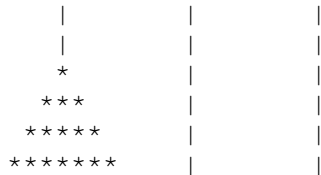


[Contenido](#)

APRENDA PASCAL

4G – Ejemplo N°4

Un problema clasico de recursividad, presentado en todo curso introductorio de Ciencias de la Computadora es el de Las Torres de Hanoi. En este problema hay tres estacas verticales. En la estaca de la izquierda hay una torre con forma de cono, consistente en una serie de discos o anillos. Por ejemplo, la siguiente es una torre de cuatro anillos:



Las estacas se designan de izquierda a derecha como 1, 2 y 3. El problema consiste en mover una torre (de cualquier altura) desde la estaca 1 a la 3. En el proceso, no se puede poner un disco mayor encima de uno menor, y sólo un disco (el disco de la posición superior) puede ser movido en culaquier momento. El problema parece trivial, y lo es para uno o dos discos. Para un disco solo, simplemente se mueve de la estaca 1 a la 3. Para dos discos, se debe mover el disco superior a la estaca 2, luego 1 a 3, y finalmente el disco más pequeño de 2 a 3.

El problema es más complejo para tres o más discos. Para tres discos, se debe mover 1 a 3, luego 1 a 2, luego 3 a 2. Esto crea una torre de 2 pisos en la estaca 2. Luego se debe mover 1 a 3. Ahora hay que mover la torre de dos pisos de la estaca 2 : 2 a 1, 2 a 3 y 1 a 3.

Su misión, si usted decide aceptarla, es escribir un programa usando un procedimiento recursivo para resolver las Torres de Hanoi para cualquier número de discos. Primero pregunte al usuario la altura de la torre original. Luego imprima paso por paso las instrucciones para mover cada disco de una estaca a otra. Por ejemplo, para tres discos la salida debería ser:

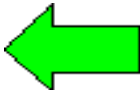
```
1 to 3
1 to 2
3 to 2
1 to 3
2 to 1
2 to 3
1 to 3
```

Como se dijo al principio dela lección de recursividad ([lección 4E](#)), la recusividad es uno de los temas más difícil de comprender. Mucha gente mirará este problema y lo encontrará ridiculamente fácil. Otros, pasarán momentos dificles con el. Sin embargo, una vez que hayan pasado el obstaculo de comprender recursividad, el problema resultará muy fácil de resolver.

Entonces, si a usted le gustan los desafíos, deje de leer exactamente aquí. Si usted tiene alguna dificultad, siga leyendo la siguiente pista.

Pista: el problema, como todo problema de recursividad, se reduce haciendose simple con cada paso. ¿Recuerda el problema de tres discos? Usted primero creó una torre de dos discos en la estaca 2, que lo ayudó a mover el disco inferior de la estaca 1 a la 3. Luego, movió los discos de la torre de dos pisos a la estaca 3. Con cuatro discos es lo mismo. Primero se debe crear una torre de tres discos en la estaca 2, luego mueva el disco mayor a la estaca 3 y mueva la torre de 3 discos a la estaca 3. ¿Cómo crear una torre de tres discos? Simple. Nosotros ya sabemos como mover una torre de tres discos desde la estaca 1 a la 3. Ahora hay que moverla de la estaca 1 a la 2, luego pase el disco más grande a la estaca 3 , y por último mueva la torre de tres discos de la estaca 2 a la 3. En este procedimiento completo, debemos actuar pensando que el disco mayor no existe. Simplemente utilice la solución del problema de tres discos, cambiando los números según corresponda.

Si usted aun no está seguro de cómo proceder, relea la [Lección 4E – Recursividad](#). Este ejemplo le dará una idea de la estructura basica de una función recursiva. Buena suerte!



[Lección anterior](#)



[Continuar](#)



[Contenido](#)

APRENDA PASCAL

4Ga - Solución a Torres de Hanoi

```
(* Author:      Tao Yue
   Date:        13 July 2000
   Description:
       Solves the Towers of Hanoi
   Version:
       1.0 - original version
*)

program TowersofHanoi;

var
    numdiscs : integer;

(*****)

procedure DoTowers (NumDiscs, OrigPeg, NewPeg, TempPeg : integer);
(* Explanation of variables:
   Number of discs -- number of discs on OrigPeg
   OrigPeg -- peg number of the tower
   NewPeg -- peg number to move the tower to
   TempPeg -- peg to use for temporary storage
*)
begin
    (* Take care of the base case -- one disc *)
    if NumDiscs = 1 then
        writeln (OrigPeg, ' ---> ', NewPeg)
    (* Take care of all other cases *)
    else
        begin
            (* First, move all discs except the bottom disc
               to TempPeg, using NewPeg as the temporary peg
               for this transfer *)
            DoTowers (NumDiscs-1, OrigPeg, TempPeg, NewPeg);
            (* Now, move the bottommost disc from OrigPeg
               to NewPeg *)
            writeln (OrigPeg, ' ---> ', NewPeg);
            (* Finally, move the discs which are currently on
               TempPeg to NewPeg, using OrigPeg as the temporary
               peg for this transfer *)
            DoTowers (NumDiscs-1, TempPeg, NewPeg, OrigPeg)
        end
    end;
end;

(*****)

begin    (* Main *)
    write ('Please enter the number of discs in the tower ==> ');
    readln (numdiscs);
    writeln;
    DoTowers (numdiscs, 1, 3, 2)
end.    (* Main *)
```



[Lección anterior](#)



[Continuar](#)



[Contenido](#)

APRENDA PASCAL

5A – Tipos de Datos Numerados

Usted puede declarar sus propios tipos de datos ordinales. Esto se hace en la sección `types` de su programa:

type

identificador_del_tipo_de_datos = especificacion_del_tipo_de_datos;

Una forma de hacer esto es creando un tipo de datos numerados. La especificación del tipo de datos numerados tiene la sintaxis:

(identificador1, identificador2, ... identificadorn)

Por ejemplo, si usted quiere declarar los meses del año, debería hacer un `type`:

`type`

```
MonthType = (January, February, March, April, May, June,
             July, August, September, October, November,
             December);
```

Usted puede declarar una variable:

`var`

```
Month : MonthType;
```

Usted puede asignarle cualquier valor numerado a la variable:

```
Month := January;
```

Todas las funciones ordinales son validas en los tipos numerados.

`ord(January) = 0`, and `ord(December) = 11`.

Los tipos numerados son internos a un programa -- nunca pueden ser leídos o escritos a un archivo de texto.

Usted debe leer los datos de entrada y convertirlos a tipo numerado. El identificador usado en un `type` (como `January`) no puede usarse en otro `type`.

El principal propósito de tipos de datos numerados es ayudar al programador a referirse a los datos con nombres con significado. Algunas veces, cuando el tipo de datos no satisface, es mas fácil usar un tipo de datos numerado que mantener el dato entrado como una cadena de caracteres.



[Lección anterior](#)



[Continuar](#)



[Contenido](#)

APRENDA PASCAL

5B - Subrangos

Un tipo de datos subrango se lo define en terminos de otro tipo de datos ordinales.

La especificación del tipo es:

valor_menor .. *valor_mayor*

donde *valor_menor* < *valor_mayor* y los dos valores estan dentro del rango de otro tipo de datos ordinal.

Por ejemplo, usted puede querer declarar los días de la semana asi como los días laborales de la semana:

```
type
  DiasDeLaSemana = (Domingo, Lunes, Martes, Miercoles, Jueves,
                    Viernes, Sabado);
  DiasLaborales  = Lunes..Viernes;
```

Usted también puede usar subrangos para tipos de datos ordinales como `char` e `integer`.



[Lección anterior](#)



[Continuar](#)



[Contenido](#)

APRENDA PASCAL

5C – Arreglos Unidimensionales

Suponga que usted desea leer 5000 enteros y hacer algo con ellos. ¿Cómo guarda los enteros? Podría usar 5000 variables:

aa, ab, ac, ad, ... aaa, aab, ... aba, ...

o podría usar un arreglo.

Un arreglo contiene varios espacios, todos del mismo tipo, para guardar datos. Se puede mencionar cada espacio de almacenamiento mediante el nombre del arreglo y subíndices.

La definición del tipo es:

type

nombre_del_tipo = arreglo [tipo_numerado] of otro_tipo_de_dato;

El tipo de dato puede ser cualquiera, aún otro arreglo. El tipo numerado se especifica dentro de corchetes o mediante tipo de datos numerados predefinidos. En otras palabras:

type

enum_type = 1..50;

arraytype = array [enum_type] of integer;

es equivalente a

type

arraytype = array [1..50] of integer;

Las cadenas de caracteres se manejan del siguiente modo:

type

String = packed array [0..255] of char;

Se usa algún carácter de terminación para indicar el final de la cadena de caracteres. Normalmente es el carácter nulo (0). La especificación packed significa que el arreglo debe ser comprimido para ocupar la menor cantidad de memoria. Esto permite imprimir todo el arreglo entero a la vez en vez de un carácter por vez. En Turbo/Borland Pascal hay incluido un tipo de datos string.

Los arreglos son útiles para guardar grandes cantidades de datos para su posterior uso en el programa.

Trabajan especialmente bien con bucles for-to, porque el índice del for-to puede ser utilizado como subíndice del arreglo. Por ejemplo para leer 50 números:

type arraytype = array[1..50] of integer;

y

var myarray : arraytype;

for count := 1 to 50 do

read (myarray[count]);

Los subíndices de los arreglos van entre corchetes [].

myarray[5] := 6;



[Lección anterior](#)



[Continuar](#)



[Contenido](#)

APRENDA PASCAL

5D – Arreglos Multidimensionales

Se pueden tener arreglos de dimensiones multiples:

type

```
datatypeidentifier = array [enum_type1, enum_type2] of datatype;
```

La coma separa las dimensiones. Para hacer referencia a un dato del arreglo:

```
a [5, 3]
```

Los arreglos bidimensionales son comunes en la programación de juegos de tableros. Un tablero de ta te ti tendría las siguientes declaraciones de tipos y variables:

type

```
StatusType = (X, O, Blank);
```

```
BoardType = array[1..3,1..3] of StatusType;
```

var

```
Board : BoardType;
```

Se debería comenzar el tablero con:

```
for count1 := 1 to 3 do
```

```
  for count2 := 1 to 3 do
```

```
    Board[count1, count2] := Blank;
```

Se puede ir a mayores dimensiones. Sin embargo, yo no se para que usted necesitaría un arreglo de dimensión 50. Incluso el espacio, de acuerdo a la teoría de las cuerdas, solo tiene 10 o 26 dimensiones!



[Lección anterior](#)



[Continuar](#)



[Contenido](#)

APRENDA PASCAL

5E - Estructuras

El tipo de variable record permite declarar estructuras. Si se desea información de una persona tal como nombre, edad, ciudad, provincia, código postal, se puede declarar una estructura:

```
TYPE  
TypeName = record  
identifierlist1 : datatype1;  
...  
identifierlistn : datatypen;  
end;
```

Por ejemplo:

```
type  
    InfoType = record  
        Name : string;  
        Age : integer;  
        City, State : String;  
        Zip : integer;  
    end;
```

Cada uno de los identificadores Name, Age, City, State, and Zip se denominan campos. Se puede acceder a un campo dentro de la variable con:

VariableIdentifier.FieldIdentifier

La variable y el nombre del campo están separados por un punto. Si usted va a trabajar con una estructura por mucho tiempo y no tiene ganas de tipear el nombre de la variable una y otra vez, puede usar sólo el identificador del campo mediante:

```
WITH RecordVariable DO  
BEGIN  
...  
END;
```

Ejemplo:

```
WITH Info DO  
    BEGIN  
        Age := 18;  
        ZIP := 90210;  
    END;
```



[Lección anterior](#)



[Continuar](#)



[Contenido](#)

APRENDA PASCAL

5F - Punteros

Un puntero es un tipo de dato que almacena una dirección de memoria. Para acceder al dato almacenado en esa dirección de memoria, se debe ver a que dirección hace referencia el puntero.

Para declarar un puntero se debe especificar a dónde debe apuntar. El tipo de dato al que va a apuntar va precedido del símbolo (^). Por ejemplo, si se crea un puntero a un entero:

```
type
PointerType = ^integer;
```

También se pueden declarar variables de tipo puntero.

Antes de acceder a un puntero se debe crear espacio de memoria para el. Esto se hace con:

```
New (PointerVariable);
```

Para acceder al dato en la dirección de memoria a la que apunta el puntero se debe usar el símbolo ^ luego del nombre de la variable. Por ejemplo, si *PointerVariable* fuera declarada como tipo *PointerType* se le podría asignar un valor a la posición de memoria a la que apunta el puntero mediante:

```
PointerVariable^ := 5;
```

Al terminar de utilizar el puntero, se debe liberar la memoria reservada. De otro modo, cada vez que se ejecuta el programa, éste reserva más y más memoria hasta que la computadora no disponga de más. Para liberar la memoria reservada se utiliza el comando **Dispose**:

```
Dispose(PointerVariable);
```

Se puede asignar un puntero a otro puntero. Sin embargo, note que ya que el puntero no contiene el valor de un dato sino su ***dirección***, una vez que se modifica el dato de la dirección apuntada por el puntero, cuando se use el otro puntero, también se referirá a la dirección del dato modificado. Además, si se libera la memoria reservada para uno de los punteros, el otro apuntará a una dirección sin sentido.

¿Para qué son útiles los punteros? ¿Por qué no se puede usar un entero en el ejemplo anterior, en vez de usar un puntero a un entero? La respuesta es que en esa situación, claramente no se necesita un punteo. El uso de punteros es para la creación de estructuras dimensionadas dinámicamente. Si se necesita almacenar muchos ítems de un tipo de datos, se puede usar un arreglo. Sin embargo, el arreglo tendrá una dimensión predefinida. Si no se tiene suficiente espacio, puede pasar que no se puedan acomodar todos los datos. Si por las dudas se define un arreglo de dimensiones enormes, se está utilizando mucha memoria cuando a veces esa memoria no se va a usar.

Una estructura dinámica, por otro lado, ocupa solamente tanta memoria como la que va a usar. Lo que se hace es crear un tipo de dato que apunte a una estructura. La estructura tendrá un puntero como uno de sus campos. Por ejemplo, usando esa estructura se pueden implementar pilas y colas:

```
type
  PointerType = ^RecordType;
  RecordType = record
    data : integer;
    next : PointerType;
  end;
```

Cada elemento apunta al siguiente. Para saber cuándo ha terminado la cadena de registros de la estructura, al siguiente campo se le asigna el valor **nil**.



[Lección anterior](#)



[Continuar](#)



[Contenido](#)

APRENDA PASCAL

Aprinda Pascal – Palabras Finales

This concludes my informal course in Pascal. Optional assignment -- program a checkers game (known in British English as draughts) in Pascal for two players. This program should check for illegal moves and should also force jumps when possible (that's in the the official rules of checkers -- if you **can** make a jump, you **must** make a jump). For a further challenge, add an option for computer play.

If you will be writing DOS programs which need mouse support, here is [mouse support code](#). This was written as a Turbo Pascal unit. If you will be writing Windows or Macintosh programs, this unit will not be necessary since those operating systems have their own mouse support routines.

If you're using [Free Pascal](#), you may want to look at the [units page](#) to find units written by other people that you may find useful in your own programs.

If you'd like to learn more about Pascal, you can go to [The Open Directory's Pascal Tutorials category](#). This has links to several more tutorials which may teach other topics that are not covered in *Learn Pascal*.

Good luck in your future Pascal endeavors!



[Lección anterior](#)



[Contenido](#)