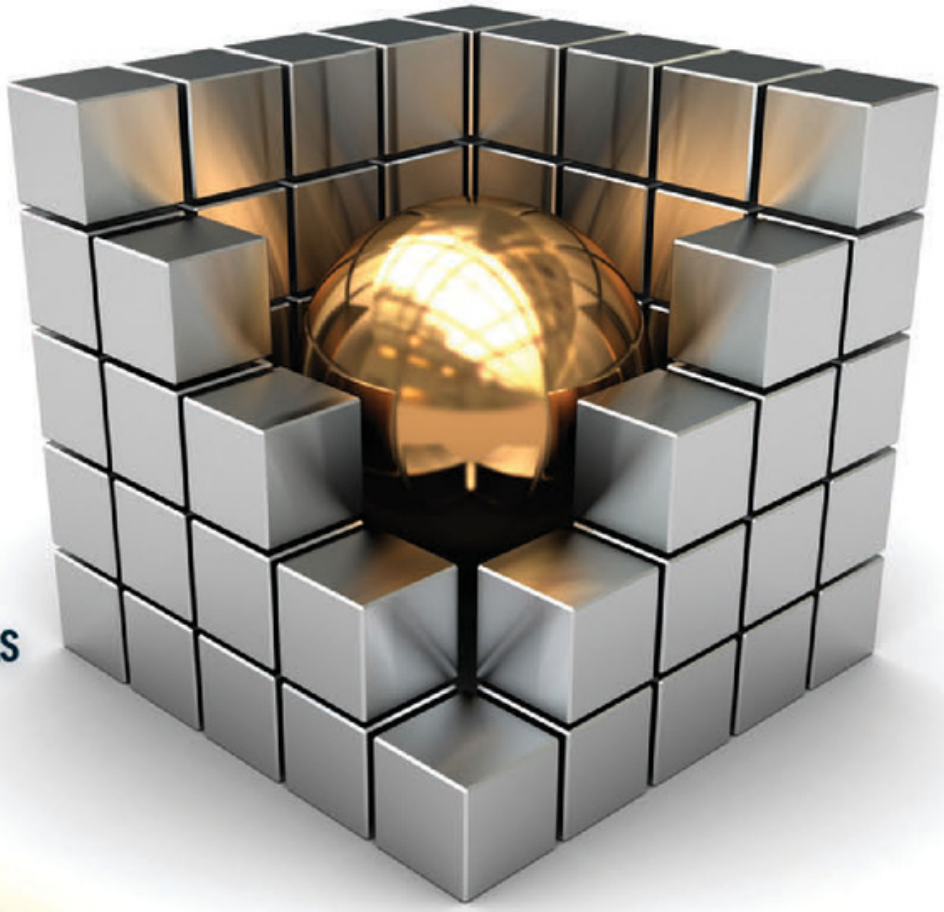
USERS

INCLUYE PROYECTOS EN VISUAL BASIC Y C++

INTRODUCCIÓN A LA PROGRAMACIÓN

MANUAL DEL DESARROLLADOR



METODOLOGÍAS, ANÁLISIS Y DISEÑO DE UN SISTEMA

DESARROLLO DE APLICACIONES: CARACTERÍSTICAS Y OBJETIVOS

PROGRAMACIÓN LÓGICA PARA CUALQUIER LENGUAJE

APLICACIONES PARA ESCRITORIO, WEB Y MÓVIL

CREACIÓN DE INTERFACES GRÁFICAS

por JUAN CARLOS CASALE

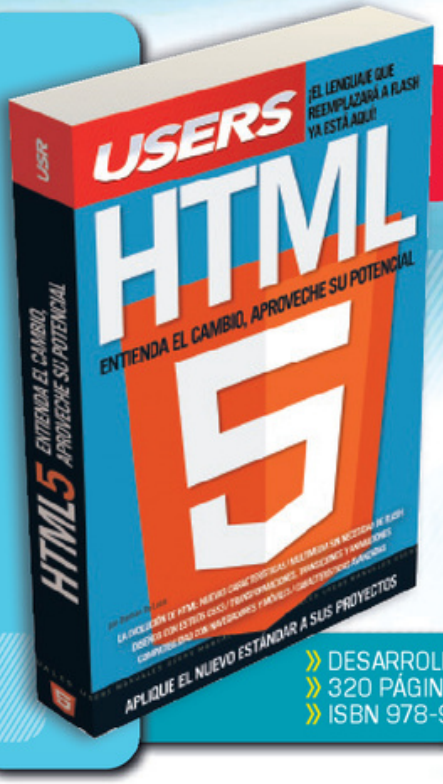
APRENDA A PROGRAMAR SIN CONOCIMIENTOS PREVIOS

RU RedUSERS

CONÉCTESE CON LOS MEJORES LIBROS DE COMPUTACIÓN

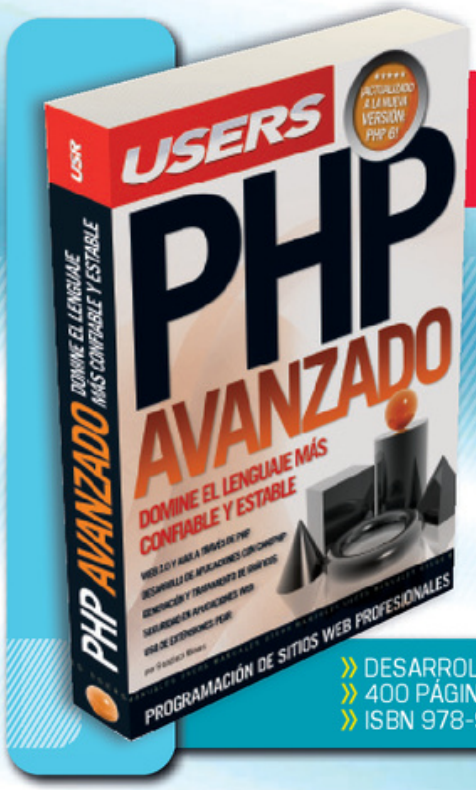
DOMINE EL LENGUAJE LÍDER EN APLICACIONES CLIENTE-SERVIDOR

» DESARROLLO
» 320 PÁGINAS
» ISBN 978-987-1773-97-8



ENTIENDA EL CAMBIO, APROVECHE SU POTENCIAL

» DESARROLLO
» 320 PÁGINAS
» ISBN 978-987-1773-79-4



PROGRAMACIÓN DE SITIOS WEB PROFESIONALES

» DESARROLLO
» 400 PÁGINAS
» ISBN 978-987-1773-07-7

LAS MEJORES PRÁCTICAS PARA EL ÉXITO PROFESIONAL

» DESARROLLO / MICROSOFT
» 352 PÁGINAS
» ISBN 978-987-1857-38-8

LLEGAMOS A TODO EL MUNDO VÍA OCA* Y DHL**

MÁS INFORMACIÓN / CONTÁCTENOS

usershop.redusers.com +54 (011) 4110-8700 usershop@redusers.com

* SÓLO VÁLIDO EN LA REPÚBLICA ARGENTINA // ** VÁLIDO EN TODO EL MUNDO EXCEPTO ARGENTINA

INTRODUCCIÓN A LA PROGRAMACIÓN

APRENDA A PROGRAMAR SIN CONOCIMIENTOS PREVIOS

por Juan Carlos Casale

RedUSERS

TÍTULO: Introducción a la programación

AUTOR: Juan Carlos Casale

COLECCIÓN: Manuales USERS

FORMATO: 17 x 24 cm

PÁGINAS: 384

ISBN 978-987-1857-69-2

Casale, Juan Carlos

Introducción a la programación. - 1a ed. - Buenos Aires : Fox Andina; Dálaga, 2012.

384 p. ; 24x17 cm. - (Manual users; 235)

ISBN 978-987-1857-69-2

1. Informática. I. Título

CDD 005.3

ANTES DE COMPRAR

EN NUESTRO SITIO PUEDE OBTENER, DE FORMA GRATUITA, UN CAPÍTULO DE CADA UNO DE LOS LIBROS EN VERSIÓN PDF Y PREVIEW DIGITAL. ADEMÁS, PODRÁ ACCEDER AL SUMARIO COMPLETO, LIBRO DE UN VISTAZO, IMÁGENES AMPLIADAS DE TAPA Y CONTRATAPA Y MATERIAL ADICIONAL.

RedUSERS
COMUNIDAD DE TECNOLOGIA

redusers.com

Nuestros libros incluyen guías visuales, explicaciones paso a paso, recuadros complementarios, ejercicios, glosarios, atajos de teclado y todos los elementos necesarios para asegurar un aprendizaje exitoso y estar conectado con el mundo de la tecnología.

LLEGAMOS A TODO EL MUNDO VÍA OCA* Y DHL**

*SÓLO VÁLIDO EN LA REPÚBLICA ARGENTINA // **VÁLIDO EN TODO EL MUNDO EXCEPTO ARGENTINA

usershop.redusers.com // usershop@redusers.com

Juan Carlos Casale

Nacido en Salta capital, norte argentino, se trasladó a Córdoba capital para estudiar Análisis de Sistemas de Computación. Allí concretó sus estudios y continúa capacitándose hasta el día de hoy.

Iniciado en la Informática por una gran influencia y motivación de su hermano mayor, creció avocándose al hardware, a las metodologías para el manejo de equipos y al ámbito de la programación en general.

Actualmente, es Analista de Sistemas y Administrador de Empresas, con aspiraciones a la Licenciatura en Tecnologías de la Educación. También es docente de Informática y coordinador de área en laboratorios de Informática del Colegio Universitario IES Siglo 21, ubicado en Córdoba capital. Además, ha editado distintos textos interactivos de estudio para la institución en donde se desempeña.

E-mail: johncasale@hotmail.com

Dedicatoria

A mi hermano mayor, Walter, con quien compartí mi primera PC, y me brindó siempre su apoyo incondicional.

Agradecimientos

A mí amada compañera de vida, Cecilia, por acompañarme durante todo el camino de desarrollo de este libro.

A Matías Iacono, por confiar en mí y dar mis referencias; además de ser un gran modelo para seguir en la Informática.

A mi súper editora Belén, con quien formamos un gran equipo y mantuvimos siempre el optimismo.

Prólogo

Cuando tenía diez años, mi hermano llegó a casa con unas cajas y yo, sinceramente, no entendía nada de lo que veía. En ese momento, la computadora no era más que un "futuro juguete"; y pensar que hoy constituye mi herramienta de trabajo y, también, mi juguete...

Después de haber quemado dos fuentes de alimentación y de sufrir algunos errores con las máquinas que tuvimos, empecé a interesarme más en su funcionamiento y a pensar qué cosas realmente productivas podía hacer con estos aparatos. Desde entonces, tomé la decisión de dedicarme a ellos, estudiando y considerando la Informática como mi vocación. De a poco fui aprendiendo sobre el desarrollo de aplicaciones –por aquellas épocas, C++–, incursionando y rompiendo cosas del hardware en varios equipos.

Al recibirme, se abrieron puertas que nunca había imaginado. La docencia se presentó frente a mí, y desde entonces, encontré mi nuevo don, que es capacitar. Hasta el día de hoy, soy feliz dando clases y aprendiendo de mis alumnos. Es un gran orgullo verlos crecer y superar obstáculos.

Nunca pensé en ser capacitador, motivador y, mucho menos, autor de libros. Hay oportunidades en la vida que nos sorprenden, y está en nosotros tomarlas o no. Mi consejo: es preferible alimentar nuestra experiencia de pruebas y errores y no desmotivarnos si las cosas no salen como lo esperábamos, ya que el verdadero fracaso sería no haberlo intentado.

A lo largo del desarrollo, nos encontraremos con distintos inconvenientes que nos dejarán alguna enseñanza para seguir probando. Espero que esta obra sea una buena base para iniciarse en el mundo de la programación. Hoy en día, contamos con variada tecnología y muchos medios a los cuales podemos dirigirnos; está en nosotros tomar las oportunidades que ofrece el libro y continuar instruyéndonos en el desarrollo de aplicaciones.

Juan Carlos Casale

El libro de un vistazo

En este libro encontraremos todas las bases necesarias para iniciarnos en el desarrollo de programas informáticos y, así, crear nuestro primer software. A lo largo de esta obra, iremos aprendiendo la lógica de la programación a partir de modelos prácticos que facilitarán la visualización y comprensión de los temas.

*01 DESARROLLO DE APLICACIONES

Por qué deseamos realizar una aplicación de software y cuál es su funcionamiento interno. En este primer capítulo, conoceremos los ámbitos en donde podemos aplicar los desarrollos de software, qué precisamos tener en cuenta a la hora de desarrollar y qué hay detrás de las aplicaciones informáticas.

*02 INICIO DE UN DESARROLLO

En este capítulo veremos las metodologías que se utilizan en el mercado del software para llevar adelante proyectos de programación. También trabajaremos sobre el análisis funcional, el ciclo de vida de un software y el diseño necesario para iniciar la programación de aplicaciones.

*03 INGRESO AL MUNDO DE LA PROGRAMACIÓN

Empezaremos por la base de la programación, constituida por el pseudocódigo, es decir, el lenguaje humano que nos permite hacer "pensar" a una máquina. En este capítulo veremos las nomenclaturas que se utilizan para escribirlo y, así, determinar el funcionamiento interno de un desarrollo de software.

*04 PRIMER PROYECTO EN VISUAL BASIC

En este capítulo pondremos en práctica lo aprendido en pseudocódigo, adentrándonos ya en un lenguaje de programación que es reconocido por su facilidad de uso. Veremos un entorno de desarrollo y las características del lenguaje en sí, para así comenzar con nuestra primera aplicación.

*05 PRIMER PROYECTO EN C++

Sumando una experiencia diferente al lenguaje estudiado en el capítulo anterior, trabajaremos con C++ en un entorno de desarrollo diferente. De esta forma, podremos reconocer las características más importantes que encierra este lenguaje y seguiremos confeccionando pequeñas aplicaciones.

*06 ESTRUCTURA DE DATOS EN LA PROGRAMACIÓN

En este capítulo veremos algunas de las estructuras de datos más utilizadas en la programación de cualquier lenguaje. Aquí repasaremos las nociones de: tipos, listas, colas y pilas; y trabajaremos en el desarrollo lógico de la programación.

*07 NORMAS GENERALES EN LAS INTERFACES GRÁFICAS

A la hora de utilizar diferentes dispositivos o medios que nos permitan mostrar nuestro desarrollo, debemos tener en cuenta ciertas pautas útiles. Estas nos permitirán la confección y el diseño de una interfaz funcional y armónica desde el lenguaje de programación Visual Basic.

*08 ALMACENAR INFORMACIÓN EN ARCHIVOS

Para ir dando un cierre a los temas vistos a lo largo del libro, aprenderemos a almacenar datos en un archivo. De esta forma, cerraremos la generación de nuestras primeras aplicaciones en un lenguaje de programación.

INFORMACIÓN COMPLEMENTARIA

A lo largo de este manual podrá encontrar una serie de recuadros que le brindarán información complementaria: curiosidades, trucos, ideas y consejos sobre los temas tratados. Para que pueda distinguirlos en forma más sencilla, cada recuadro está identificado con diferentes iconos:


CURIOSIDADES E IDEAS


ATENCIÓN


DATOS ÚTILES Y NOVEDADES


SITIOS WEB

RedUSERS
MEJORA TU PC

La red de productos sobre tecnología más importante del mundo de habla hispana

Libros
Desarrollos temáticos en profundidad

Coleccionables
Cursos intensivos con gran desarrollo visual

Revistas
Las últimas tecnologías explicadas por expertos

RedUSERS
redusers.com
Noticias al día
downloads, comunidad

Newsletters
El resumen de noticias
que te mantiene actualizado
Regístrate en redusers.com

RedUSERS PREMIUM
premium.redusers.com
Nuestros productos en versión digital con contenido ampliado y a precios increíbles

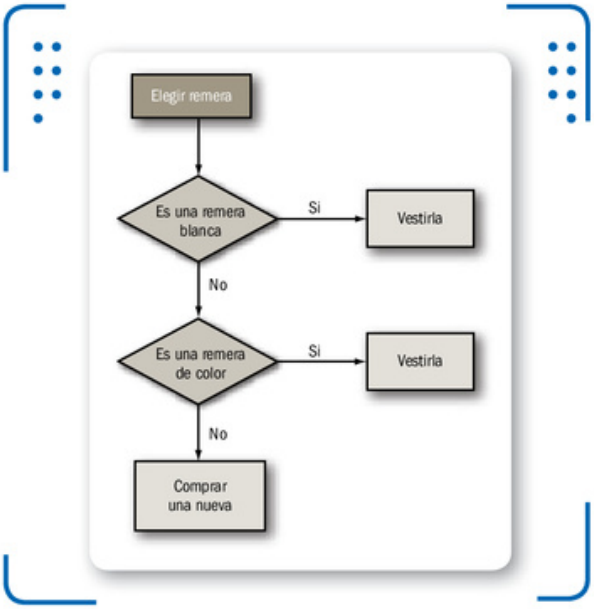
Usershop
usershop.redusers.com
El ecommerce de RedUSERS, revistas, libros y fascículos a un clic de distancia. Entregas a todo el mundo

Contenido

Sobre el autor 4
Prólogo 5
El libro de un vistazo 6
Información complementaria 7
Introducción 12

*01 Desarrollo de aplicaciones

Desarrollo de aplicaciones 14
Propósitos para aprender a desarrollar 15
Tipos de aplicaciones 18
Aplicaciones web 18
Aplicaciones de escritorio 19
Aplicaciones móviles 20
Interpretación de las aplicaciones 20



Entrada/Proceso/Salida 25
Primeras tareas de un desarrollador 26
Qué es un algoritmo 27
Qué es un lenguaje de programación 28
Etapas en la resolución de un problema 34
Resumen 35
Actividades 36

*02 Inicio de un desarrollo

Metodologías de desarrollo 38
Tipos de metodologías 39
Ciclo de vida de un desarrollo 41
Funciones del ciclo de vida 42
Roles profesionales 43
Modelos de ciclo de vida 44
Generalidades sobre metodologías 50
Análisis de sistema 51
Relevamiento 53
Diseño de un sistema 57
Diagrama de casos de uso 58
Prototipos 62
Implementación del desarrollo 65
Prueba o testing de aplicaciones 65
Capacitación y formación del usuario 66
Resumen 67
Actividades 68

*03 Ingreso al mundo de la programación

La lógica de un humano y de una máquina 70
Pseudocódigo: el lenguaje humano 71
Normas para el pseudocódigo 71
Qué son y cómo se usan las variables 75
Cómo se utilizan los operadores 80
Todo tiene un orden en la programación 94
Estructuras de control 94
Tipos de datos estructurados 109

Vector 110
Matriz 114
Utilizar funciones y procedimientos 120
Ámbito de las variables 121
Funciones 121
Procedimientos 127
Resumen 129
Actividades 130

*04 Primer proyecto en Visual Basic

Lenguajes de programación 132
Tipos de lenguajes 132
Interfaces gráficas 134
Nomenclatura en pseudocódigo y lenguajes de programación 138
Lenguaje de programación: Microsoft Visual Basic 140
Creación de proyectos 140
Qué son y cómo se usan las variables 146
Cómo se utilizan los operadores 154
Todo tiene un orden en la programación 159
Tipos de datos estructurados 168
Uso de controles básicos 175
Resumen 187
Actividades 188

*05 Primer proyecto en C++

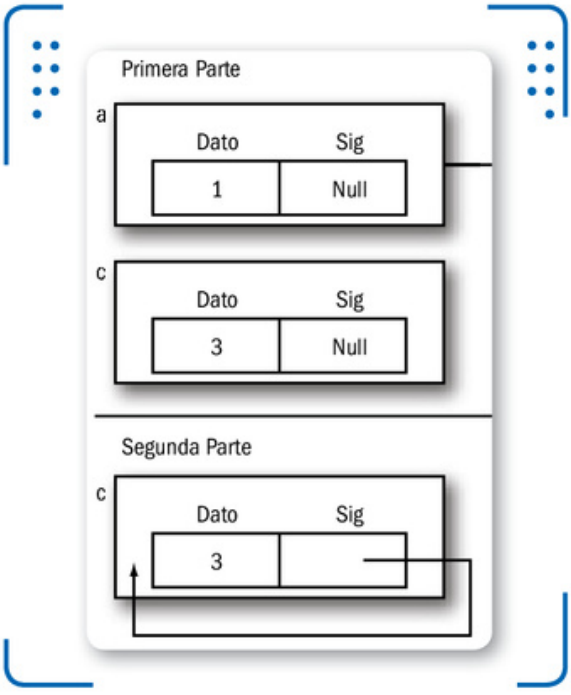
IDE SharpDevelop 190
Funcionamiento del entorno de programación 192
Lenguaje de programación: C++ 195
Nombre de espacios 196
Conceptos básicos del código 199
Primera aplicación en C++ 201
Manejo de datos en C++ 203
Tipos de datos 203
Declaración de variables 204
Inicializar variables 207
Formas de declarar constantes 211
Cómo se utilizan los operadores 213
Asignación (=) 213
Operadores aritméticos 214
Asignación compuesta 215
Aumentar y disminuir 216
Operadores relacionales y de igualdad 216
Operadores lógicos 217
Operador condicional (?) 219
Operador coma (,) 220

Operadores bitwise o bit a bit 220
Tipo de operador de conversión explícita 221
Operador sizeof () 222
Precedencia de los operadores 222
Interactuar con el usuario 224
Cout - salida estándar 224
Cin – entrada estándar 226
Todo tiene un orden en la programación 230
Estructura condicional 230
Estructuras selectiva (switch) 232
Estructuras repetitivas (loop) 234
Salto de declaraciones 239
Datos estructurados: arrays 243
Manejo de arrays 243

Sintaxis de inicialización 244
Arrays multidimensionales 246
Recorrer arrays 248
Resumen 249
Actividades 250

*06 Estructura de datos en la programación

Tipos de estructuras 252
Datos simples y estructurados 253
Estructuras dinámicas y estáticas 256
Estructuras dinámicas y punteros 257
Lista 268



Listas enlazadas 269
Listas doblemente enlazadas 293
Pila 312
Crear una pila 313
Insertar en una pila (push) 316
Eliminar en una pila (pop) 318
Listar los elementos de una pila 320
Buscar elementos en una pila 320
Cola 323
Crear una cola 325
Insertar en una cola 327
Eliminar elementos de una cola 328
Listar los elementos de una cola 330
Buscar elementos en una cola 332
Resumen 333
Actividades 334

*07 Normas generales en las interfaces gráficas

Normas de diseño de interfaz 336
Interfaces de usuario: evolución y estado del arte actual 337
Fundamentos del diseño de interfaz 345
Interfaces de escritorio/web/móvil 349
Componentes usuales 349
Componentes usuales - visuales 355
Confección de interfaces en Visual Basic 357
Resumen 365
Actividades 366

*08 Almacenar información en archivos

Almacenar en archivo de texto (FileSystem) 368
Resumen 375
Actividades 376

* Servicios al lector

Índice temático 378

Introducción

Ante un mundo en constante innovación y descubrimiento, es el desarrollo del software el que nos permite gestionar la tecnología en hardware. Estos avances nos ofrecen un amplio abanico de posibilidades, en donde podemos desempeñarnos como desarrolladores, ya sea en la industria de los video juegos, las aplicaciones web, los sistemas informáticos de organizaciones, entre otros.

Si hacemos una observación diez años atrás, veremos los cambios revolucionarios que ha traído la era digital en sus avances tecnológicos, y cómo ellos han afectado en nuestra comunicación diaria. Para dominar esta variedad de dispositivos –tanto smartphones, computadoras, tablets, Smart TV, ultranotebooks, etc.–, necesitamos desarrollar programas que nos permitan interactuar con ellos.

Este libro está dirigido a todos aquellos que quieran iniciarse en el mundo de la programación y conocer las bases necesarias para crear su primer software. A lo largo de su contenido, nos plantearemos qué nos impulsa a comenzar un desarrollo de aplicaciones y qué partes lo constituyen. Cuando conozcamos el manejo y la confección de los programas, empezaremos a incursionar en la lógica misma de la programación y podremos movernos en cualquier tipo de lenguaje.

En definitiva, el contenido del libro no se dirige hacia un único camino, sino que se propone brindar las herramientas necesarias para que sea el lector quien elija sobre qué escenario trabajar: escritorio, web, móvil, consola de videojuegos, etc. Todas estas oportunidades que ofrece la programación fueron alimentando la pasión y la experiencia que hoy presento en esta obra. ¡Que la disfruten!

*01 –

Desarrollo de aplicaciones

Es fundamental conocer y comprender los elementos iniciales de los procesos que debemos tener en cuenta para incursionar en el mundo de la programación de aplicaciones. En este capítulo vamos a desplegar varias interfaces de soporte que utilizaremos en nuestros futuros desarrollos.

- **Desarrollo de aplicaciones 14**
  - Propósitos para aprender a desarrollar 15
- **Tipos de aplicaciones 18**
  - Aplicaciones web 18
  - Aplicaciones de escritorio 19
  - Aplicaciones móviles 20
- **Interpretación de las aplicaciones 20**
- **Primeras tareas de un desarrollador 26**
  - Qué es un algoritmo 27
- **Etapas en la resolución de un problema 34**
- **Resumen 35**
- **Actividades 36**

Desarrollo de aplicaciones

Como futuros desarrolladores, nos propondremos encontrar distintas soluciones posibles para resolver una situación mediante la confección de aplicaciones informáticas. En los siguientes párrafos, vamos a conocer el significado del desarrollo de aplicaciones, y la utilidad que nos ofrecen sus diferentes técnicas y herramientas.

En el mundo actual, todos los días nos encontramos con distintos desarrollos de aplicaciones, como, por ejemplo, el programa que controla nuestro teléfono móvil. A su vez, contamos con programas que, en tiempo real, nos permiten traducir diferentes idiomas, conectarnos a Internet, jugar, llevar un listado de lo que compramos en el supermercado registrando su código de barras y estimando el costo total, y muchas alternativas más.

> TODOS LOS ELEMENTOS ELECTRÓNICOS CONTIENEN APLICACIONES

Podemos notar que algunas aplicaciones son más básicas, y otras, más complejas. Si bien es posible considerar el teléfono móvil como un aparato complejo, el desarrollo de aplicaciones también impacta en otros elementos de uso cotidiano, tales como las heladeras inteligentes, el programa del microondas, las alarmas, y otros. El mundo en su totalidad se rige por programas desarrollados mediante algún lenguaje de programación. Todos los elementos electrónicos, en menor o mayor grado, contienen aplicaciones específicas para cumplir su misión.

Una definición que podemos encontrar en primera instancia sobre el desarrollo de una aplicación es: confeccionar, probar y buscar errores de un programa informático. Dicho programa va a soluciona una situación o problema comúnmente llamado "modelo de negocio", que

NUEVOS DISPOSITIVOS

A medida que la tecnología avanza, vamos incorporado "inteligencia" en distintos dispositivos. Es así que algunas marcas conocidas fabrican heladeras, aspiradoras, lavadoras y secarropas que incluyen comunicación por medio de WiFi, y otras, incluso, tienen cámaras. Todo esto permite que el usuario controle el dispositivo por medio de un software instalado en su dispositivo móvil.

puede ser, por ejemplo, cuando nuestra empresa necesita llevar un inventario de productos. Para poder confeccionar un programa informático, precisamos emplear un lenguaje de programación que nos permita realizar la prueba o búsqueda de errores.

Figura 1. Con este software, el sistema de manejo de los alimentos permite a los usuarios conocer qué hay en la heladera, dónde está cada producto y cuál es su fecha de caducidad.

Propósitos para aprender a desarrollar

Cuando nos proponemos aprender a desarrollar y programar aplicaciones o sistemas, lo hacemos para cubrir determinadas necesidades, ya sean personales o de terceros, y así obtener un ingreso económico a cambio de nuestro trabajo.

Uno de los pasos fundamentales que debemos efectuar antes de comenzar es aprender la **programación lógica**. Esto es importante porque, si bien los lenguajes de programación tienen sus particularidades, las soluciones lógicas son analizadas de un solo modo. De esta manera, conocer este tema claramente nos permitirá migrar a todos los lenguajes que queramos.

Aprender a desarrollar aplicaciones nos ofrece muchas posibilidades, ya que podremos realizar programas en cualquier plataforma, ya sea para la Web, Windows, Linux o Macintosh; incluso, para móviles, televisión inteligente, etc. El propósito principal es tener la base lógica de programación, y luego elegir cuál es el lenguaje en el que deseamos poner nuestro mayor esfuerzo. Puede ser el que esté latente en el mercado, uno específico de un área (como para los trabajos científicos) o, simplemente, aquel en el que nos sintamos más cómodos para trabajar.

Al adquirir estos conocimientos, podremos tomar cualquier modelo de negocio o problema funcional de una organización, y resolverlo mediante la programación de una aplicación.

Resolver problemas: metas y objetivos

Nuestra tarea principal será realizar una aplicación para resolver un problema en particular, o tal vez lo hagamos solo por diversión. Por ejemplo, podemos crear un programa para llevar en nuestro teléfono móvil una agenda que nos informe los días de estreno de nuestras series favoritas de televisión. También podemos aplicarlo en el trabajo, para agilizar la toma de decisiones y digitalizar la información referida al desempeño de los empleados. Ambos son modelos de negocios distintos que plantean un problema, y nosotros debemos encontrar una solución. Estas necesidades pueden surgir desde distintos ámbitos:

- **Personal**: realizar pequeñas o amplias aplicaciones para un fin que nos beneficie. Por ejemplo: elegir una aplicación que nos indique el consumo de Internet en nuestro teléfono móvil o programar una página web personal.
- **Empresarial**: realizar sistemas informáticos, partes o módulos que tenemos que programar; incluso, arreglar un código que haya sido

TRABAJO FREELANCE

En la actualidad existe una amplia variedad de sitios dedicados a presentar ofertas laborales de modo freelance, permitiendo establecer contacto con las compañías y los recursos humanos. Algunos ejemplos de ellos son: **www.smartise.com**, **www.trabajofreelance.com**, **www.mercadoprofesional.com**, **www.stratos-ad.com** y **http://pcmasmas.com**.

confeccionado por otro. Por ejemplo: utilizar nuestros conocimientos para mejorar un sistema de inventario o realizar una página web para una organización que cuenta con un módulo de ventas online.

Tengamos en cuenta que el ámbito empresarial es más duro, ya que requiere seguir ciertas pautas y criterios que veremos en los próximos capítulos. En cambio, cuando las metas son personales, podemos dedicarnos a desarrollar de manera **freelance**, siendo nosotros mismos el sustento económico, y quienes organizamos las entregas y los horarios de trabajo. Una meta personal debería ser aprender cada día más para acrecentar nuestra experiencia, y saber que, por medio de errores y pruebas, iremos optimizando nuestro trabajo.

> EL PLANTEO DE METAS ES UN PUNTO EXCLUYENTE EN EL DESARROLLO DE APLICACIONES

Las metas empresariales son estrictas y, en general, nos afectan, ya que, por ejemplo, nos imponen un límite de tiempo específico que debemos cumplir. Dentro del desarrollo de aplicaciones, una meta empresarial que debe influir en nuestros objetivos personales es absorber los conocimientos del grupo de trabajo, para luego aplicarlos a los nuevos desafíos que vayamos afrontando más adelante.

El planteo de metas es un punto excluyente en el desarrollo de aplicaciones, porque tener en claro hacia dónde queremos llegar nos motivará a nivel personal a seguir investigando, buscando y probando. Al mismo tiempo, nos ayudará a plantearnos los objetivos buscados sobre los desarrollos a realizar. De esta forma, algo que parece tan sencillo como plantearse una meta y conocer los objetivos nos permitirá organizar y optimizar el desarrollo.

LENGUAJE BASIC

BASIC originalmente fue diseñado en 1964 como un medio para facilitar el desarrollo de programas de computación a estudiantes y profesores que no se dedicaran específicamente a las ciencias. Su aparición como herramienta de enseñanza estaba diseñada para la primera computadora personal. Con los años, el lenguaje se popularizó e influyó en gran medida en otros, como Visual Basic.

En resumen, a la hora de desarrollar una aplicación que resuelva un modelo de negocio o problema, ya sea personal o empresarial, debemos tener presentes nuestras metas, evaluar si el alcance del desarrollo es a corto o largo plazo, y establecer claramente cuáles serán nuestros objetivos a seguir.

Hasta aquí hemos visto cómo el desarrollo de las aplicaciones nos servirá para crear o modificar aquellos programas que permitirán realizar una o varias actividades. En los próximos capítulos, conoceremos cómo debería conformarse un equipo de desarrollo, en función del planteo de soluciones a problemas, metas y objetivos.

Tipos de aplicaciones

En el mercado informático actual, nos encontramos con diferentes soportes de hardware que albergan variados tipos de aplicaciones, ya sea exclusivas de Internet, del sistema operativo o de un aplicativo en particular. Así como antes comenzamos a formar el concepto de desarrollo de una aplicación, ahora vamos a reforzarlo haciendo un repaso de las aplicaciones existentes, de modo de tener una idea gráfica de qué podemos considerar para nuestro trabajo.

Aplicaciones web

Las aplicaciones web son herramientas muy comunes en organizaciones que desean ampliar las fronteras de sus modelos de negocios o, simplemente, alcanzar la autogestión para empleados, alumnos, docentes, etcétera.

QUÉ ES EL SOFTWARE LIBRE

Si bien muchas veces el término **software libre** se confunde con **freeware**, es importante tener en cuenta que se trata de conceptos distintos. La diferencia principal reside en que este último no tiene como condición ser gratuito. La denominación de "libre" se debe a que son programas de **código abierto (Open Source)**, y es en ese punto en donde se encuentra la esencia de su libertad.

Algunas páginas web que poseen una programación agradable son:

- Portal de bancos (todos tienen autogestiones completas, donde se pueden hacer extracciones, movimientos, pagos, etc.):
  - **www.macro.com.ar**
  - **www.santanderrio.com**
  - **www.hsbc.com**
  - **www.icbc.com.cn**
- Portal educativo (permite registrar usuarios y contraseñas, publicar contenidos, efectuar búsquedas, etc.):
  - **www.educ.ar**
- Portal de juegos (permite registro de usuarios, subir listas de puntuaciones, compartir comentarios, etc.):
  - **www.armorgames.com**

Hay una amplia variedad de sitios web destinados a distintos rubros, como puede ser el automotriz, en donde es posible personalizar o armar autos a gusto, elegir colores, definir agregados, etc. Esto nos demuestra la variedad de trabajo que se puede realizar en los desarrollos para la Web.

Aplicaciones de escritorio

Las aplicaciones de escritorio son aquellas que funcionan sobre un sistema operativo de PC (computadora personal) o notebook. Los desarrollos en este ámbito también son enormes, y podemos encontrarnos con algunos muy costosos, utilizados por grandes empresas; y con otros gratuitos y útiles que pueden servirnos para diferentes tareas. Por ejemplo, podemos dar un vistazo a:

- **www.softpedia.com**
- **www.softonic.com**

> EXISTEN DISTINTAS VERSIONES DE PRUEBA QUE SON GRATUITAS PARA NUESTRO EQUIPO

Veremos que muchos de estos programas cuentan con un tipo de distribución llamado **trial**. Se trata de una instalación de prueba, generalmente por un máximo de 30 días a partir de su instalación, que suele tener funcionalidades limitadas. Otras versiones de prueba gratuitas pueden ser **shareware** o **freeware**, que podemos instalar y utilizar en los equipos que queramos.

Debido a la amplia variedad que existe en el mercado, en este libro vamos a presentar las aplicaciones más destacadas de este momento. Una de ellas es **Adobe Photoshop**, reconocida por sus famosos retoques de imágenes. Frente a esto, es bueno saber que existen otras alternativas gratuitas a las que podemos recurrir, como es el caso de **GIMP**. Para conocer más sobre este soft, podemos ingresar en **www.gimp.org**.

Aplicaciones móviles

Son aplicaciones que se utilizan en equipos móviles, como teléfonos celulares o tabletas. Suelen ser muy similares a las de escritorio, ya que permiten realizar las mismas tareas, aunque el ingreso de datos es táctil o por voz. Para visualizar algunos ejemplos, podemos visitar la página del mercado de Android, donde hay una infinidad de opciones gratuitas y pagas: **https://play.google.com/store**.

Interpretación de las aplicaciones

Hasta aquí hemos realizado una introducción referida a lo que podemos encontrar en el mercado del software; ahora aprenderemos cómo es el funcionamiento interno de un programa y cuáles son los aspectos más importantes que debemos tener en cuenta, para así conocer el trasfondo de lo que vamos a desarrollar.

El proceso de funcionamiento puede ser sencillo si lo trabajamos con ejemplos, pero se vuelve más complejo en el proceso lógico real. No obstante, todas las aplicaciones suelen tener la misma estructura de ejecución. Para comenzar, no ahondaremos en el hardware implicado, pero sí en la interpretación de un programa por medio del equipo informático. Debemos tener en cuenta la manera en que un sistema de cómputos electrónico interpreta la información y cómo nosotros, futuros desarrolladores, la vemos. Todo comienza por los famosos bits de datos. Un bit representa la unidad de medida más pequeña en información digital, y tiene dos estados: 0 o 1; generalmente, el 0 se representa como cerrado (o negativo) y el 1 como abierto (o positivo).

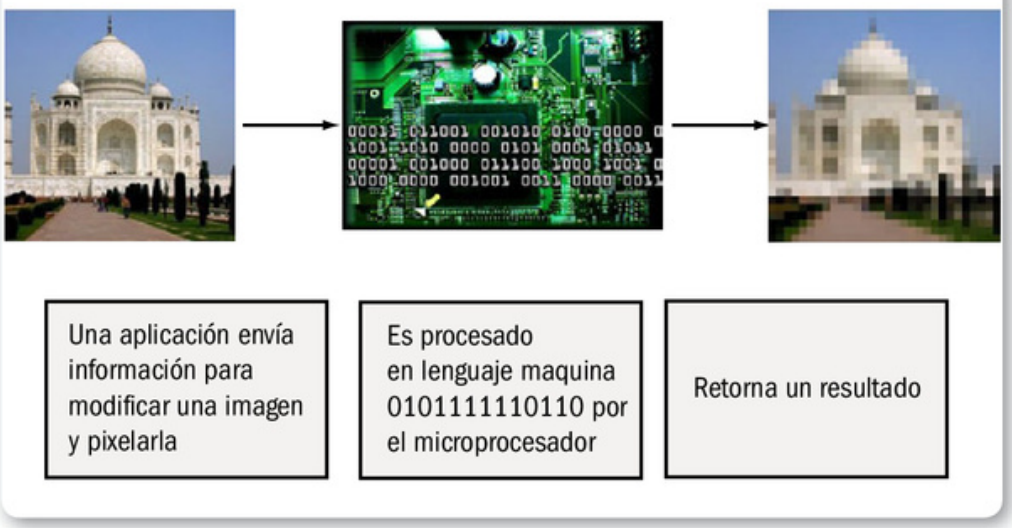


Figura 2. Una forma sencilla de ver el funcionamiento del programa en la computadora.

En la **Figura 2** se muestran diferentes agentes que debemos tener en cuenta en el uso de un dispositivo informático y la comunicación entre los equipos. Es importante saber que podemos utilizar nuestro hardware –ya sea una PC, un teléfono móvil o una tableta– gracias a un software base o aplicación base llamado sistema operativo. Sobre este sistema es posible instalar diferentes aplicativos, como: paquetes de oficina, procesadores de texto, planillas de cálculo, juegos, herramientas de desarrollo, de diseño, y otros. El sistema nos permite el uso de nuestro hardware y, además, actúa como intermediario entre la aplicación y los usuarios.

Hasta aquí hemos visto que las aplicaciones se comunican con nuestro hardware por medio de un protocolo binario (0 y 1), conocido

LENGUAJE C

C es un lenguaje de programación creado en 1972, orientado a la implementación de sistemas operativos, concretamente, UNIX. Es el lenguaje de programación más popular para crear software de sistemas y apreciado por la eficiencia de su código. Fue desarrollado, originalmente, por programadores para programadores.

como **lenguaje de máquina**. Para entender la comunicación cotidiana que existe entre los usuarios y las aplicaciones, podemos decir que, en la actualidad, la interacción se da por medio de interfaces gráficas; es decir, de una manera visual, a través de iconos, colores y formas.

Sin embargo, no podemos limitarnos a decir que la única interfaz es la **visual**, porque existen diferentes aplicaciones que utilizan varios de nuestros sentidos: tacto, oído e, incluso, olfato.

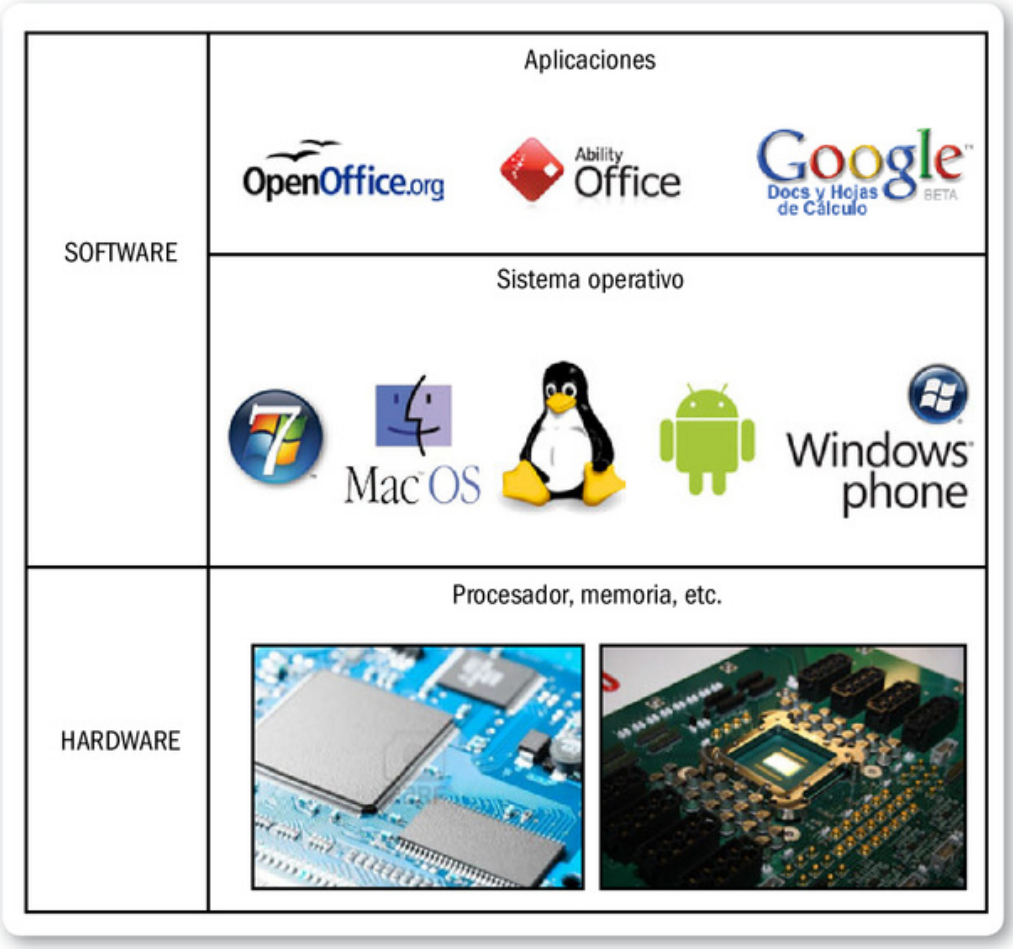


Figura 3. Esquema sencillo referido a la comunicación de los elementos en un sistema informático.

Si nos detenemos un momento a revisar qué es en realidad una interfaz y cuáles son sus diferentes significados, encontraremos que existe una gran variedad que nos involucra como usuarios. A continuación, veamos algunos ejemplos.

LOS SENTIDOS

TIPO	DESCRIPCIÓN	DISPOSITIVOS
Tacto	En la actualidad, es muy frecuente el uso de los dedos para interactuar con dispositivos para su manejo y uso.	- Smartphones - Tabletas - Mesas Touch www.youtube.com/watch?v=xQzSP26vcfw - Dispositivos para no videntes: http://www.yankodesign.com/2009/06/15/touchphone-for-the-blind/
Vista	Uno de los sentidos más involucrados para interactuar con los diferentes dispositivos, en especial, para la salida de información. Existen dispositivos especiales para ciertas discapacidades, que pueden usarse para interactuar en las pantallas como puntero de mouse.	- Monitores - Televisores - Webcams - Vinchas que reconocen los movimientos oculares: www.youtube.com/watch?v=A92WNMd46VI
Movimiento psicomotriz	Implica interactuar con nuestro cuerpo en diferentes dispositivos. Los videojuegos son los principales impulsores de esta interfaz.	- Kinect - Wii - PS Move
Olfato	En desarrollo constante, se intenta involucrar este sentido en algún dispositivo para percibir aromas por medio de una aplicación.	No hay nada concreto desarrollado

Tabla 1. Aquí se pueden ver los sentidos humanos que se involucran en diferentes componentes tecnológicos.

FREEWARE

Podemos considerar freeware a todo aquel programa que se distribuya de manera gratuita, sin ningún costo adicional. Uno de los grandes ejemplos en este rubro es la suite de navegador, cliente de correo y noticias de Mozilla, como así también el navegador y las herramientas de Google. Una página que podemos visitar al respecto es www.freewarehome.com.

Figura 4. La botonera de un ascensor, como tipo de interfaz táctil, nos permite enviar la orden al dispositivo.

Tomando este tema como una pequeña introducción acerca de cómo los sentidos se involucran en distintas interfaces que podemos utilizar, en este caso nos dedicaremos a interfaces gráficas y táctiles, las que más utilizamos en nuestra vida cotidiana, al menos por ahora.

Figura 5. Otro ejemplo cotidiano de interfaz táctil es el panel de un microondas, que utilizamos para programar la cocción.

Para concluir con este tema, podemos decir que la interacción humana con un dispositivo electrónico siempre va a llevarse a cabo por medio de una interfaz. A su vez, para que esto ocurra, la interfaz debe contener algún software que interactúe con el hardware del dispositivo, y así nos permita obtener el resultado deseado.

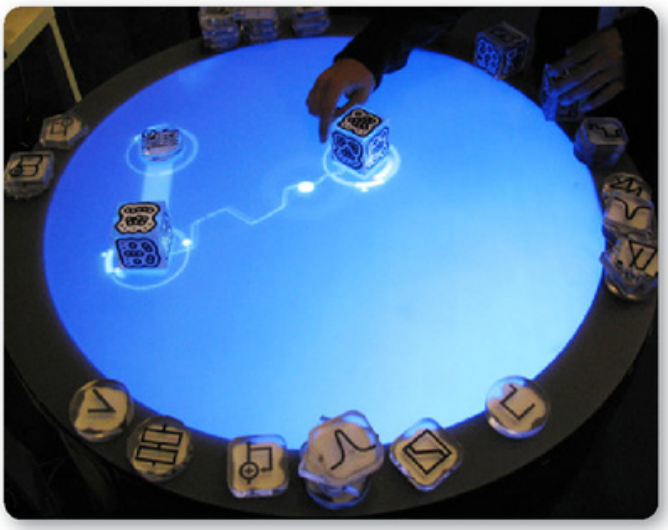


Figura 6. La interfaz de la mesa con tablero translúcido actúa al identificar el movimiento de los objetos sobre ella.

Entrada/Proceso/Salida

La **entrada** es el ingreso o comando de datos que vamos a realizar sobre un dispositivo, como, por ejemplo: tocar la pantalla, escribir, mover el puntero del mouse, hacer el movimiento con un joystick, etc. Por lo tanto, toda entrada se hará por medio de un dispositivo, como puede ser una pantalla táctil, un teclado, una webcam o un mouse.

El **proceso** es el trabajo, la interpretación y el cálculo de la información ingresada. Esta información puede ser un movimiento del mouse, una tecla pulsada, datos para calcular enviados, y otros. Fundamentalmente, en el proceso ya entran en juego el procesador y la memoria de un dispositivo.

La **salida** es el resultado de las acciones que se efectúan sobre la información. Por lo tanto, si pulsamos el botón del mouse, se ejecutará una aplicación (pulsar el botón **Enviar** de un correo), se realizará una

¿QUÉ ES UNA INTERFAZ?

Según la RAE (Real Academia Española, www.rae.es), el término interfaz significa conexión física y funcional entre dos aparatos o sistemas independientes, como puede suceder entre una persona y un dispositivo electrónico. Un ejemplo claro de esto se ve en la **Figura 4**, en donde utilizamos una botonera como interfaz para indicarle a un ascensor el piso al que queremos ir.

acción en un juego (como disparar), se devolverá el resultado de un cálculo, se ejecutará un video, y otras opciones más.

Este proceso de retroalimentación nos dará los mismos resultados, presionando ya sea uno o varios botones del teclado.

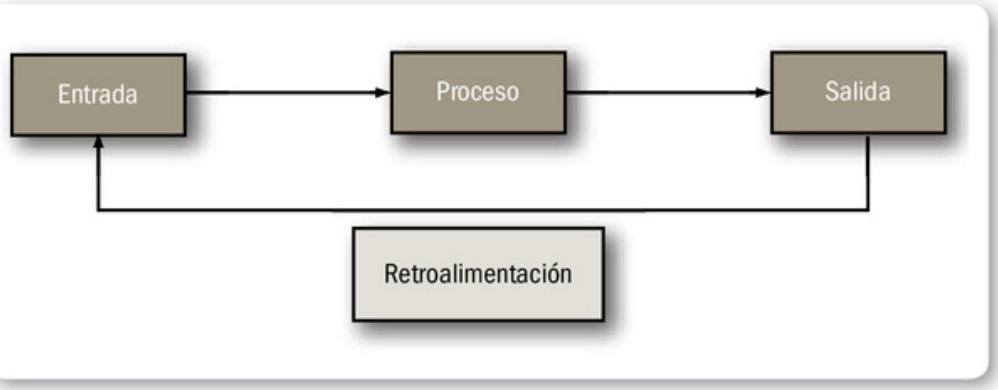


Figura 7. El proceso de retroalimentación comienza con el ingreso de la información y concluye con la acción emitida por el dispositivo.

Primeras tareas de un desarrollador

Hasta este punto, hemos visto que la interacción con dispositivos electrónicos se presenta por medio de interfaces. Estas, a su vez, cuentan con un software que traduce nuestras acciones a un lenguaje máquina reconocido por el hardware, con lo cual se obtiene un resultado. Para lograr esto, como desarrolladores es importante que conozcamos la manera de darle al equipo informático las indicaciones necesarias. En este libro aprenderemos a confeccionarlas por medio del estudio de la lógica de programación, y a plasmarlas en líneas de código de un software específico para diagramar y tipear.

A continuación, desarrollaremos dos conceptos fundamentales que debemos tener bien en claro durante el desarrollo: **algoritmia** y **lenguajes de programación**. Una vez que los dominemos, podremos lograr que el software cumpla con todas nuestras indicaciones.

Qué es un algoritmo

Si bien encontraremos múltiples definiciones de lo que es un algoritmo, nosotros trabajaremos con la genérica que toma la RAE, en la que se hace referencia a un conjunto ordenado y finito de operaciones que permite hallar la solución de un problema.

Nosotros, como seres humanos, tenemos incorporado un "**algoritmo**" de decisiones. Por ejemplo, si deseamos vestir una remera, realizamos un proceso de selección de cuál o tal queremos, y terminamos por hacer la selección deseada. En un conjunto ordenado y finito de operaciones, podríamos representar, a través de un algoritmo, este proceso de selección y solución.

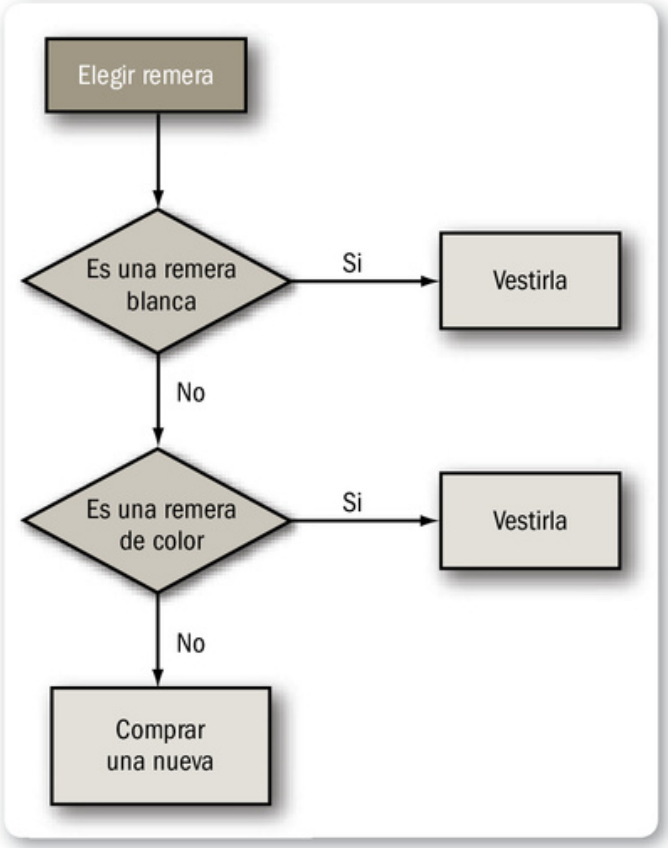


Figura 8. Proceso de selección de una remera para vestir. Se trata de una representación de algoritmos llamado diagrama de flujo.

De esta manera, podemos definir el algoritmo como una serie de pasos ordenados que debemos seguir para lograr, finalmente, la resolución de una situación o problema. En el desarrollo, para poder ejecutar una aplicación, tenemos que traducir esto a sentencias ordenadas de código que se ejecuten línea a línea.

Qué es un lenguaje de programación

Anteriormente presentamos la comunicación que existe entre un software y el hardware. Ahora vamos a conocer la comunicación que debemos establecer nosotros, como desarrolladores, frente a nuestro hardware, para lograr que este ejecute las tareas o procesos que deseamos. Para este fin, necesitaremos como herramienta primordial un **lenguaje de programación**.

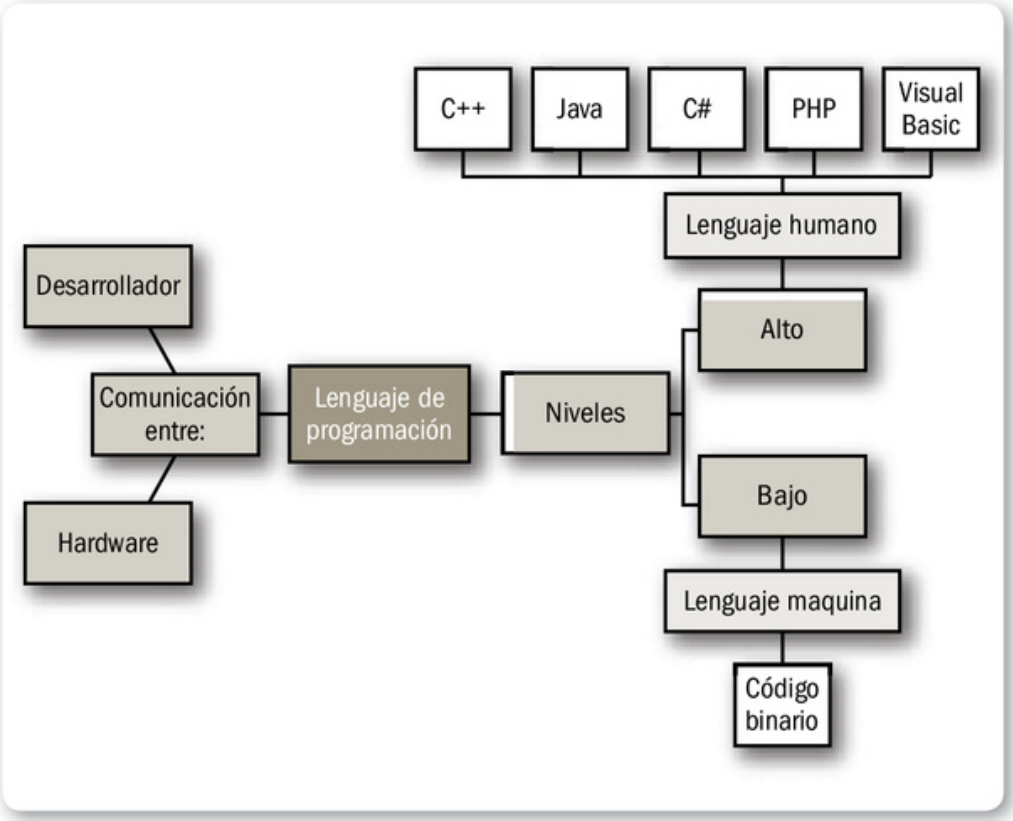


Figura 9. En este cuadro conceptual vemos la representación de los distintos lenguajes de programación.

Existen muchos lenguajes de programación que nos permiten desarrollar, por medio de un código (protocolo), sentencias algorítmicas que luego son traducidas a lenguaje máquina. Estos cumplen la función de intermediarios entre el desarrollador y el hardware.

Teniendo en cuenta esta diversidad, veremos que hay dos grupos generales. Por un lado, se encuentran los lenguajes más próximos a la arquitectura del hardware, denominados lenguajes de bajo nivel (son

más rígidos y complicados de aprender). Por otro lado, están aquellos más cercanos a los programadores y usuarios, denominados lenguajes de alto nivel (son más comprensibles para el lenguaje humano). En la **Figura 9** vemos una representación clara de este concepto.

En distintos escritos se consideran lenguajes de **bajo nivel** a algunos como: FORTRAN, ASSEMBLER y C. Como lenguajes de **alto nivel** podemos mencionar: Visual Basic, Visual C++ y Python. Si bien podemos encontrar categorizaciones más finas al respecto, que describan diferentes tipos de lenguajes, recordemos que, en términos generales, siempre se habla de lenguajes de alto nivel y de bajo nivel.

DISTINTOS NIVELES

▼ LENGUAJE	▼ DESCRIPCIÓN
Máquina	Código interpretado directamente por el procesador. Las invocaciones a memoria, como los procesos aritmético-lógicos, son posiciones literales de conmutadores físicos del hardware en su representación booleana. Estos lenguajes son literales de tareas. Ver su ejemplo en la Figura 10.
Bajo nivel	Instrucciones que ensamblan los grupos de conmutadores necesarios para expresar una mínima lógica aritmética; están íntimamente vinculados al hardware. Estos lenguajes están orientados a procesos compuestos de tareas, y la cantidad de instrucciones depende de cómo haya sido diseñada la arquitectura del hardware. Como norma general, están disponibles a nivel firmware, CMOS o chipset. Ver su ejemplo en la Figura 11.
Medio nivel	Son aquellos que, basándose en los juegos de instrucciones disponibles (chipset), permiten el uso de funciones a nivel aritmético; pero a nivel lógico dependen de literales en ensamblador. Estos lenguajes están orientados a procedimientos compuestos de procesos. Este tema se verá a continuación, en el ejemplo Código C y Basic.
Alto nivel	Permiten mayor flexibilidad al desarrollador (a la hora de abstraerse o de ser literal), y un camino bidireccional entre el lenguaje máquina y una expresión casi oral entre la escritura del programa y su posterior compilación. Estos lenguajes están orientados a objetos. Ver su ejemplo en la Figura 12.

Tabla 2. Las diferencias básicas entre los lenguajes de programación.

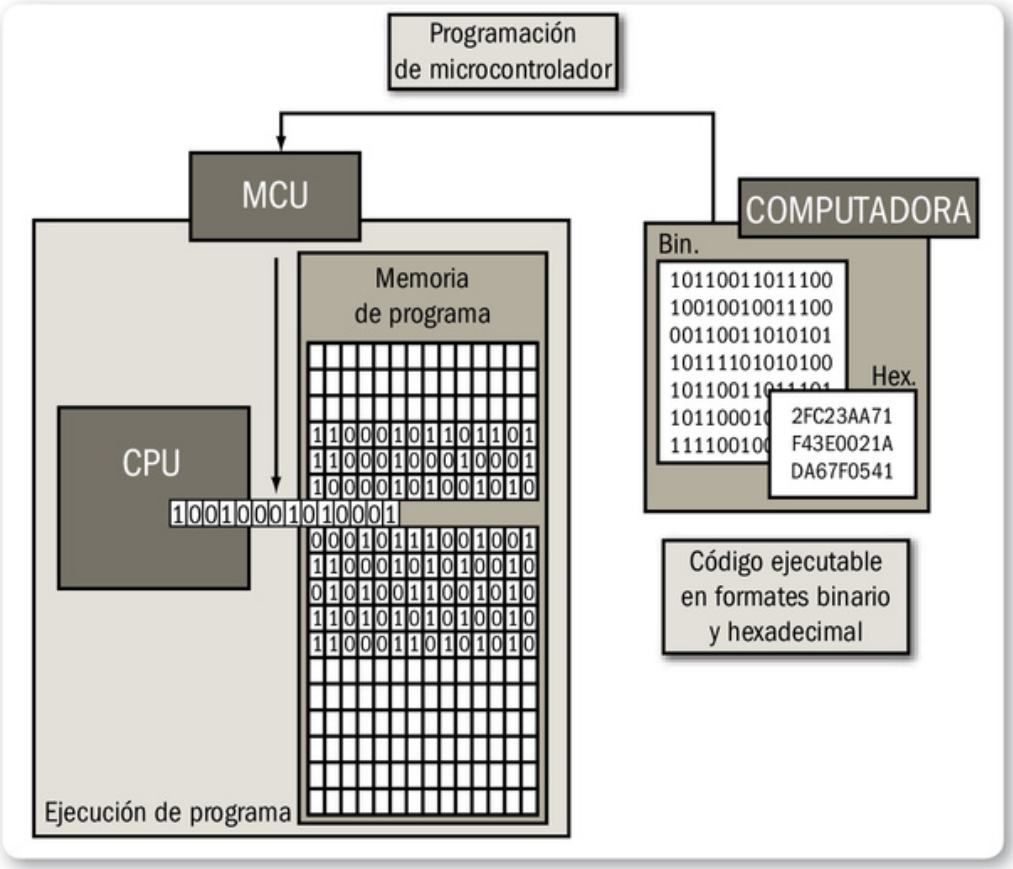


Figura 10. Interpretación general de cómo se vería la comunicación de instrucciones al procesador.

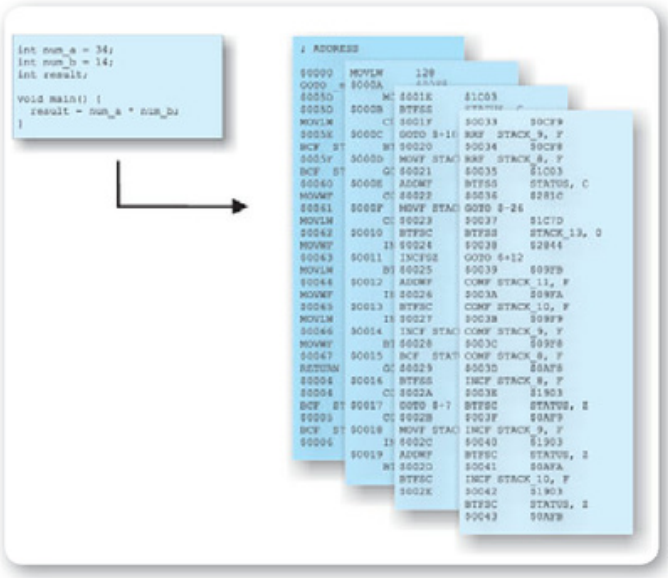


Figura 11. El código fuente de la izquierda representa el código en lenguaje C; y el de la derecha, el código en ASSEMBLER.

En los cuadros que aparecen a continuación veremos el código fuente de dos lenguajes de programación muy importantes: C y Basic. Al principio no debemos asustarnos, se trata solo de ejemplos para ir conociendo cómo se vería el desarrollo aplicado en ellos.

En el siguiente código veremos en detalle la declaración de variables requeridas para el lenguaje C.

```
C
int main()
{
 /* Declaración de variables */

 int ano,dia;
 int mes;
 int total;
 int i;
int sol;
}
```

En el lenguaje de programación de Basic, el código que aparece a continuación significa: "limpia" la pantalla de cualquier texto, luego asigna valores a las variables (como el nombre y un número) y finaliza con la impresión en pantalla del resultado.

```
BASIC
CLS
nombre1$="George Kemeny"
valor1=500
nombre2$="Eugene Kurtz"
valor2=350
PRINT nombre1$
PRINT valor1
PRINT nombre2$
PRINT valor2
```

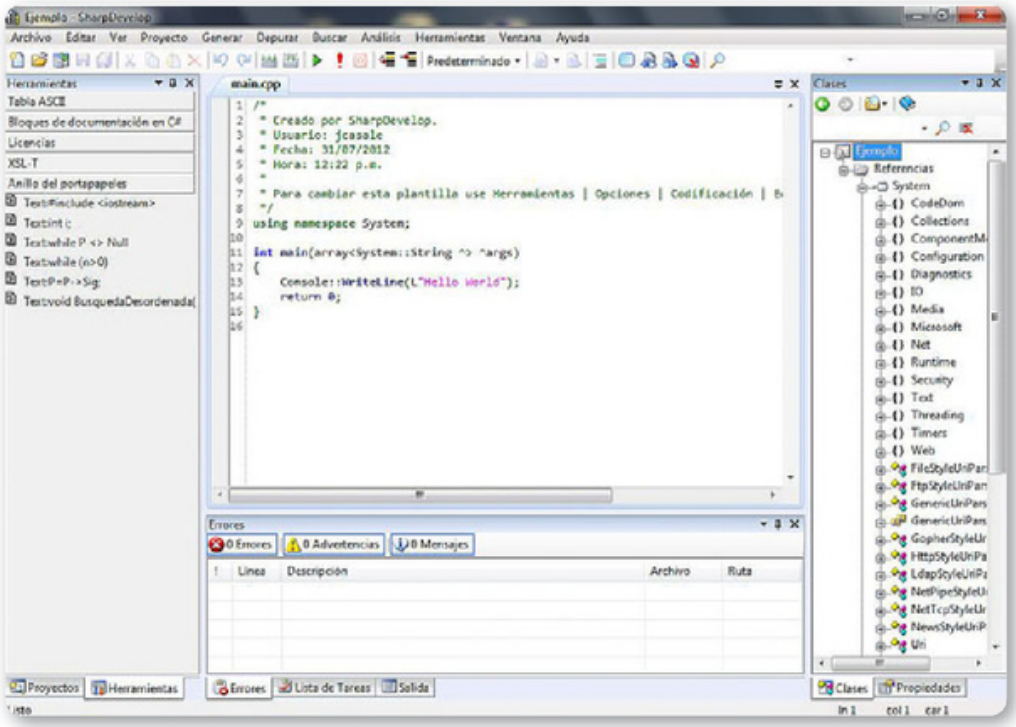


Figura 12. Ejemplo sobre el conjunto de herramientas que constituyen un entorno de desarrollo visual en C#.

Dentro de la amplia variedad de lenguajes de programación que mencionamos antes, en la **Figura 13** podremos observar algunos de los que se presentan actualmente en el mercado.

A continuación, veremos un listado que incluye otras fuentes a las que podemos acceder para conocer los rankings o tener más información sobre los distintos lenguajes de programación.

- **www.tiobe.com**
- **http://langpop.com**

DIAGRAMA DE FLUJO

Los diagramas de flujo son descripciones gráficas de algoritmos que usan símbolos conectados mediante flechas para indicar la secuencia de instrucciones. Este tipo de diagramas se utilizan para representar algoritmos pequeños. Su construcción es laboriosa y son medianamente fáciles de leer para personas que no están involucradas en la programación.

- **www.genbetadev.com/desarrolladores/ranking-de-lenguajes-de-programacion-mas-usados-en-marzo-2011**
- **www.realmagick.com/timeline-of-programming-languages**

Hasta aquí, conocimos la comunicación establecida entre un software y el hardware, y cómo debemos actuar para lograr que, por medio de algoritmos, nuestro sistema realice las tareas que nosotros deseamos. Esto es posible gracias al lenguaje de programación que elijamos, ya sea el que más nos gusta o aquel que se adapta mejor a las necesidades del mercado actual.

Más allá de la decisión que tomemos, este libro nos servirá como guía y apoyo para aprender a trabajar sobre cualquier lenguaje, ya que nos brindará las bases necesarias para comprender y saber aprovechar mejor sus diferentes características.

Posicion Jul 2012	Posicion Jul 2011	Delta en la posición	Lenguaje de programación	Valoración Jul 2012	Delta Jul 2011	Estado
1	2	↑	C	18.331 %	+1.05 %	A
2	1	↓	Java	16.087 %	-3.16 %	A
3	5	↑↑	Objective-C	9.335 %	+4.15 %	A
4	3	↓	C++	9.118 %	+0.10 %	A
5	4	↓	C#	6.668 %	+0.45 %	A
6	7	↑	(Visual) Basic	5.695 %	+0.59 %	A
7	5	↓↓	PHP	5.012 %	-1.17 %	A
8	8	=	Python	4.000 %	+0.42 %	A
9	9	=	Perl	2.053 %	-0.28 %	A
10	12	↑↑	Ruby	1.768 %	+0.44 %	A
11	10	↓	JavaScript	1.454 %	-0.79 %	A
12	14	↑↑	Delphi/Object Pascal	1.157 %	+0.27 %	A
13	13	=	Lisp	0.997 %	+0.09 %	A
14	15	↑	Transact-SQL	0.954 %	+0.15 %	A
15	25	↑↑↑↑↑↑↑↑↑↑	Visual Basic.NET	0.917 %	+0.43 %	A
16	16	=	Pascal	0.837 %	+0.17 %	A
17	19	↑↑	Ada	0.689 %	+0.14 %	A
18	11	↓↓↓↓↓↓↓	Lus	0.684 %	-0.89 %	A
19	21	↑↑	PL/SQL	0.645 %	+0.10 %	A
20	26	↑↑↑↑↑↑	MATLAB	0.639 %	+0.19 %	A

Figura 13. Algunos de los lenguajes más utilizados durante el año 2012. Fuente: **www.tiobe.com**

Etapas en la resolución de un problema

Ahora que conocemos las herramientas involucradas en el desarrollo de aplicaciones, es conveniente revisar qué tareas generales debemos considerar para llevar a cabo esta labor.

APLICAREMOS HERRAMIENTAS DE UN LENGUAJE PARA RESOLVER LA SITUACIÓN

Como seres humanos, tenemos incorporada intuitivamente la resolución de problemas cotidianos gracias a nuestra experiencia, y para intentar afrontar un inconveniente, solemos hacer un proceso rápido de selección e intentamos buscar la opción más favorable. En el ámbito laboral, y más aún en el desarrollo de aplicaciones, debemos ser cautelosos al momento de resolver alguna tarea o proceso. Por eso, nos será de gran utilidad aplicar una herramienta del lenguaje de programación que nos permita confeccionar un programa y, así, resolver dicha situación.

Si bien este esquema nos será útil para la resolución de un desarrollo sencillo, en caso de trabajar con sistemas amplios, deberemos incluir ciertas técnicas de ingeniería del software.

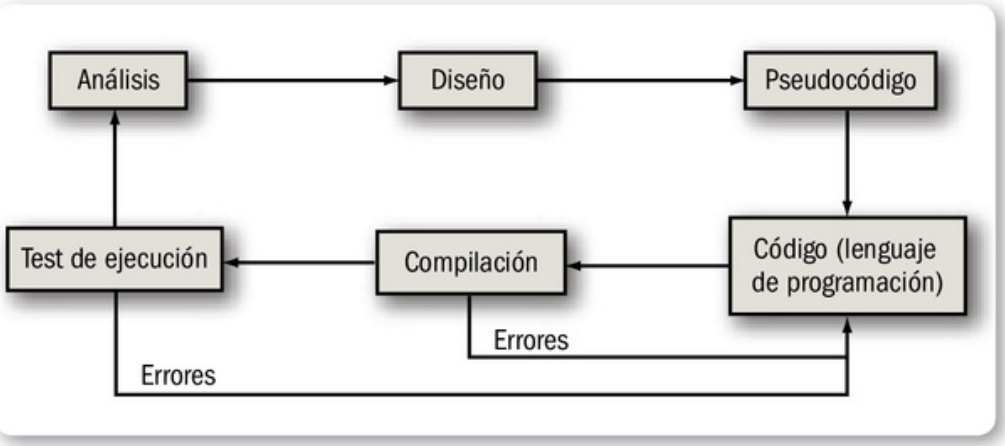


Figura 14. En este esquema aparecen las principales etapas que se encuentran involucradas durante el desarrollo de aplicaciones.

En el proceso que vemos en el gráfico de la **Figura 14**, puede suceder que debamos retroceder y volver a analizar o replantear algunas de las acciones. Revisemos los pasos expuestos en el esquema (y en los próximos capítulos veremos cómo se desarrolla una aplicación basada en él). Los siguientes aspectos son pasos que seguiremos como desarrolladores para resolver una situación:

- Analizar el problema que vamos a resolver.
- Diseñar una solución.
- Traducir la solución a pseudocódigo.
- Implementar en un lenguaje de programación todo lo analizado.
- Compilar el programa.
- Realizar pruebas de ejecución.
- Corregir los errores que haya.

RESUMEN

En este capítulo empezamos a conocer el funcionamiento del software y revisamos la comunicación que tiene el hardware con los programas intangibles y abstractos. Vimos que esta comunicación se desarrolla por medio de un lenguaje máquina, y nosotros nos comunicamos con el equipo electrónico por medio de interfaces. Gracias al lenguaje de programación, nosotros, como desarrolladores, podemos indicar las acciones que deseamos realizar a través de algoritmos.

Por último, vimos la manera en que debemos empezar a encarar la resolución de los problemas, teniendo en cuenta el lenguaje y la lógica de programación.

Actividades

TEST DE AUTOEVALUACIÓN

1	¿Qué es el lenguaje máquina?
2	¿Cuántos niveles de lenguaje de programación existen?
3	¿Qué es un lenguaje de alto nivel?
4	¿Qué es un algoritmo?
5	¿Cómo se comunica el hardware con el software?
6	¿Qué es el código binario?
7	¿Cuántos tipos de aplicaciones podemos encontrar?
8	¿Qué es un lenguaje de programación?
9	¿Qué representa en el software la entrada/proceso/salida?
10	¿Cuáles pueden ser los propósitos para realizar un desarrollo?

*02

Inicio de un desarrollo

A medida que la tecnología y la innovación en la informática progresan, algunos profesionales del ámbito consideran que es necesario seguir ciertas pautas predefinidas en el desarrollo del software, basadas en el comportamiento metódico y el intenso análisis de sistemas.

- **Metodologías de desarrollo 38**
- **Ciclo de vida de un desarrollo . 41**
  Funciones del ciclo de vida 42
- **Análisis de sistema 51**
- **Diseño de un sistema 57**
- **Implementación del desarrollo 65**
- **Prueba o testing de aplicaciones 65**
- **Capacitación y formación del usuario 66**
- **Resumen 67**
- **Actividades 68**

Metodologías de desarrollo

Debido a las múltiples maneras que existen para conceptualizar una metodología, es complicado llegar a un acuerdo para definir qué es una **metodología de desarrollo**. Sin embargo, podemos encontrar un concepto en común que la define como un **framework** utilizado para estructurar, planear y controlar el proceso de desarrollo. De este modo, las metodologías nos proveerán de una organización que aplicaremos a los diferentes proyectos de programación.

A la hora de conceptualizar una metodología, notaremos que existe una amplia variedad de enfoques a los que podemos recurrir. Para obtener una definición clara y asegurarnos de no dejar de lado ninguna cuestión importante, vamos a crear nuestro propio significado. Para eso, seleccionaremos los conceptos fundamentales que involucran a una metodología y analizaremos sus funciones:

- **Metodología**: conjunto de procedimientos, técnicas, herramientas y soporte documental que utilizan los desarrolladores a la hora de tomar las decisiones sobre el software a realizar.
- **Tarea**: actividades elementales en las que se dividen los procesos.
- **Procedimiento**: forma que se define para ejecutar la tarea.
- **Técnica**: herramienta utilizada para aplicar un procedimiento; es posible usar una o varias.
- **Herramienta**: para realizar una técnica, podemos apoyarnos en las herramientas de software que automatizan su aplicación.
- **Producto**: resultado de cada etapa.

Ahora que ya hemos comprendido estos conceptos que involucran a las metodologías de desarrollo, estamos en condiciones de analizar los distintos tipos de metodologías que existen.

FRAMEWORK

Dentro de la metodología de desarrollo, el framework es el enfoque del proceso con el que vamos a contar para realizar o utilizar un software. Se trata de una fuente de herramientas, modelos y métodos que podemos tomar y usar para efectuar distintas acciones, como, por ejemplo, dibujar una pantalla. Ejemplos: AJAX, .NET, Axis, y otras.

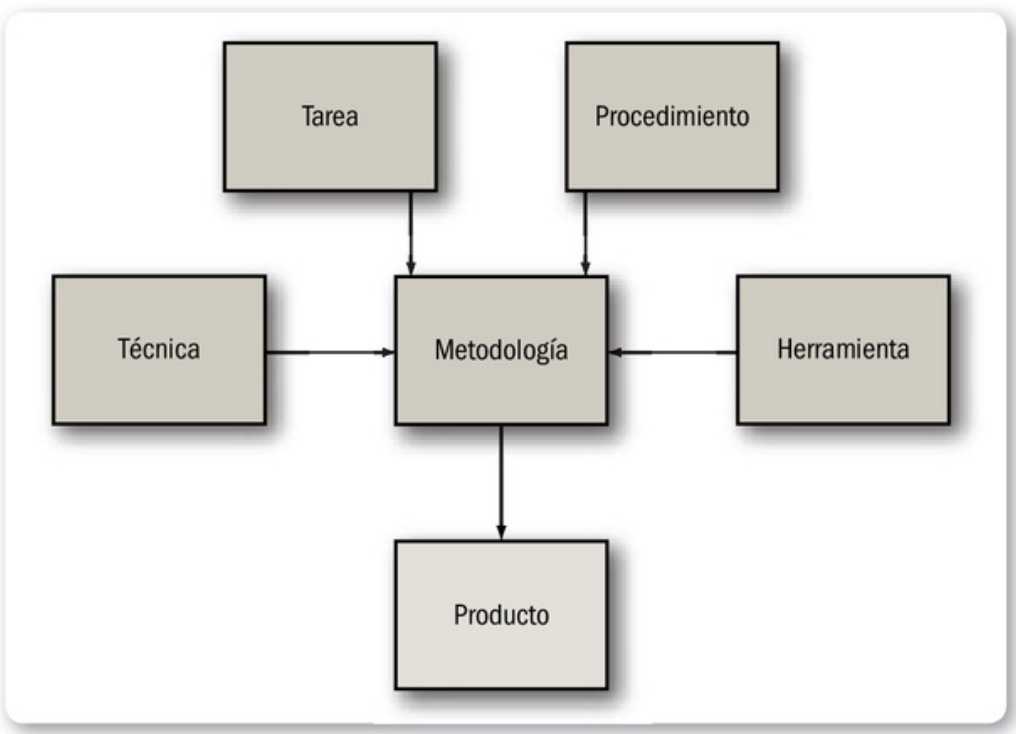


Figura 1. En la producción de un producto, cada uno de estos aspectos influye en forma directa sobre la metodología.

Tipos de metodologías

Dentro del ámbito informático, existe una gran variedad de metodologías de desarrollo. En la tabla que aparece a continuación vamos a conocer estos tipos, para que más adelante podamos decidir correctamente cuál se aplica mejor a nuestro proyecto.

METODOLOGÍAS

CLASIFICACIÓN	METODOLOGÍAS
Tradicionales/Pesadas	Cascada, Modelo V, RAD, MERISSE, METRICA, SSADM, RUP
Iterativas/Evolutivas	Prototipos, Espiral, Espiral WIN&WIN, Entrega por etapas, RUP
Ágiles	XP, SCRUM, CRISTAL, Desarrollo adaptable, Open Source, DSDM, Desarrollo manejado por rasgos, Code and Fix

Tecnología Web	OOHDM,HDM, RNA, etc.
Otras	Orientada a aspectos, Sistemas de tiempo real, Basado en componentes

Tabla 1. Tipos de metodologías en el desarrollo del software.

Dentro de todas estas categorías, nosotros vamos a enfocarnos en las más utilizadas por aquellas organizaciones que se dedican al desarrollo de las aplicaciones informáticas. Pero, a su vez, debido a la amplia variedad que existe en el ámbito informático, tendremos que evaluar cuáles son las que se aplican mejor a nuestros proyectos, para así adaptarlas y ejecutarlas según nuestras necesidades. En este análisis sobre qué metodología utilizar, es importante tener en cuenta que la elección diferirá según el país, la provincia o, incluso, el centro de enseñanza al que pertenezcamos. Desde el punto de vista humano, todos tenemos gustos y pensamientos diferentes acerca de cómo vemos nuestro entorno; por eso, la elección dependerá en gran medida de cuál sea nuestra situación económica, política y social.

Muchas veces se pone el énfasis en que las metodologías deben planificar, controlar, capturar requisitos, realizar tareas de modelado, y promover la etapa de análisis y diseño antes de proceder a la construcción del software. Pero también es importante que seamos muy detallistas en la documentación utilizada en cada una de las etapas. Este tipo de metodología se denomina **tradicional** o **pesada**.

Figura 2. Junto con el UML, constituye la metodología más utilizada para los sistemas orientados a objetos.

En contraste con estas metodologías, encontraremos las **ágiles**. Estas sostienen que el desarrollo del software debe ser considerado como un modelo incremental, donde las entregas de este sean más pequeñas, en ciclos más cortos y rápidos. De esta manera, se elimina la burocracia de la documentación que caracteriza a las metodologías tradicionales. Esto se logra gracias a una forma más cooperativa entre el cliente y los desarrolladores, por medio de una comunicación más cercana y fluida. En las metodologías ágiles, la documentación más importante está constituida por el código fuente.

> LAS METODOLOGÍAS ÁGILES APLICAN ENTREGAS MÁS CORTAS Y RÁPIDAS AL SOFTWARE

Ya vimos que, dentro del ámbito de la programación, existen distintos tipos de metodologías de desarrollo. También dimos un vistazo al concepto sobre ellas, y aprendimos que el uso de una u otra dependerá de nuestro entorno, el equipo y los recursos con los que contemos. A continuación, vamos a indagar en las cuestiones que debemos tener en cuenta a nivel profesional acerca de los desarrollos de software. Podemos ampliar más sobre las metodologías consultando otro texto de nuestra editorial: *Métodos Ágiles*, por Sebastián Priolo.

Ciclo de vida de un desarrollo

Una vez que hemos determinado la necesidad de realizar un software, es importante prestar atención a su ciclo de vida; es decir, el conjunto de **fases** por las cuales pasa la idea inicial, desde que nace hasta que el software es retirado o reemplazado.

Cuando hablamos del **nacimiento** de un proyecto, nos referimos a la idea o problema puntual que se presenta. Luego, esta irá creciendo gracias a la actualización y recaudación de información que surja, además de a su puesta a prueba cotidiana. De esta forma, el desarrollo va **madurando** hasta llegar a la **muerte** o **reemplazo** del producto.

Estas tareas o actividades que debemos desempeñar en el ciclo de vida del software suelen representarse en una serie de grandes bloques: **análisis**, **diseño**, **producción**y **mantenimiento**.

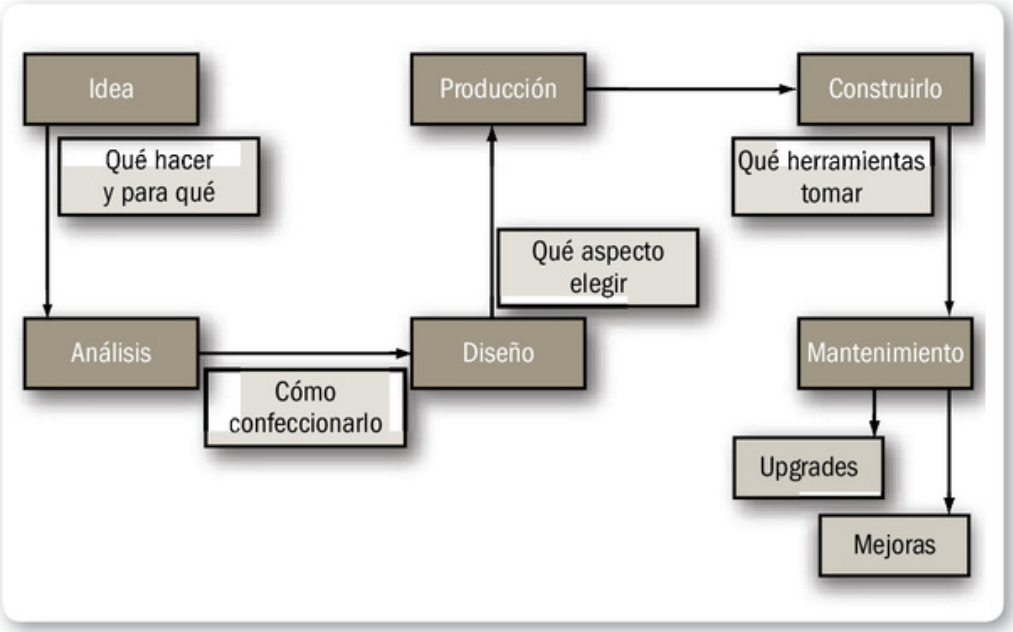


Figura 3. Bloques principales que representan las tareas por desempeñar en el ciclo de vida del software.

Funciones del ciclo de vida

Entre las funciones que debe tener un ciclo de vida, desde que nace hasta que muere, podemos destacar las siguientes:

- Determinar el orden de las fases del proceso de software.
- Establecer los criterios de transición para pasar de una fase a la otra.
- Puntualizar las entradas y salidas de cada fase.
- Describir los estados por los que pasa el producto.
- Especificar las actividades a realizar para transformar el producto.
- Definir un esquema que sirva como base para planificar, organizar, coordinar y desarrollar el proceso.

Como podemos observar, el ciclo de vida del desarrollo de un software es complejo si deseamos llevar a cabo todos los pasos que corresponden. Recordemos que, a la hora de elegir la metodología adecuada, es preciso tener en cuenta el ámbito donde lo desarrollaremos. Si es en una organización dedicada exclusivamente al desarrollo, es necesario mantener los estándares de calidad sobre los productos, y es en este punto donde entran en juego las normas ISO y otros estándares.

Roles profesionales

Dentro de cada fase que vayamos a realizar, hay distintos roles profesionales involucrados. A continuación, mencionaremos los más importantes y su característica fundamental:

- **Analista de sistema**: generalmente, puede integrarse en cualquier etapa del ciclo de vida de un software, aunque, en esencia, lo encontramos en el inicio.
- **Líder de proyecto**: es aquel que lleva a cabo la organización y el seguimiento de cada fase.
- **Arquitecto en software / Analista funcional**: son las mentes que llevarán a cabo la maquetación y el diseño, además de la documentación del proyecto.
- **Desarrollador**: se ocupa de codificar los prototipos y esquemas que le suministren en un lenguaje de programación.
- **Soporte / Tester**: brinda apoyo al cliente del software y realiza testing de las aplicaciones, lo que lleva a las pruebas exhaustivas y documentadas generalmente antes de entregar un producto.
- **Calidad**: revisa la documentación, para luego presentar los elementos necesarios a las organizaciones de normalización y calidad de los productos.

> ES MUY IMPORTANTE LA ELECCIÓN DEL MODELO Y EL ORDEN ESTABLECIDO PARA UN PROYECTO

Dentro de cada una de las etapas de un modelo de ciclo de vida, es posible definir una serie de objetivos, tareas y actividades que lo caractericen, lo que permite llevar un importante proceso administrativo. El hecho de que existan distintos modelos hace que sea tan importante su elección y el orden establecido para un proyecto determinado.

NORMAS ISO

La Organización Internacional para la Estandarización, ISO por sus siglas en inglés (*International Organization for Standardization*) es una federación mundial que establece un conjunto de reglas referidas a la calidad y la gestión continua de la norma. Su objetivo es desarrollar estándares internacionales que se apliquen a cualquier tipo de organización o actividad orientada a la producción de bienes o servicios, coordinando y unificando los usos para conseguir menores costos y una mayor efectividad.

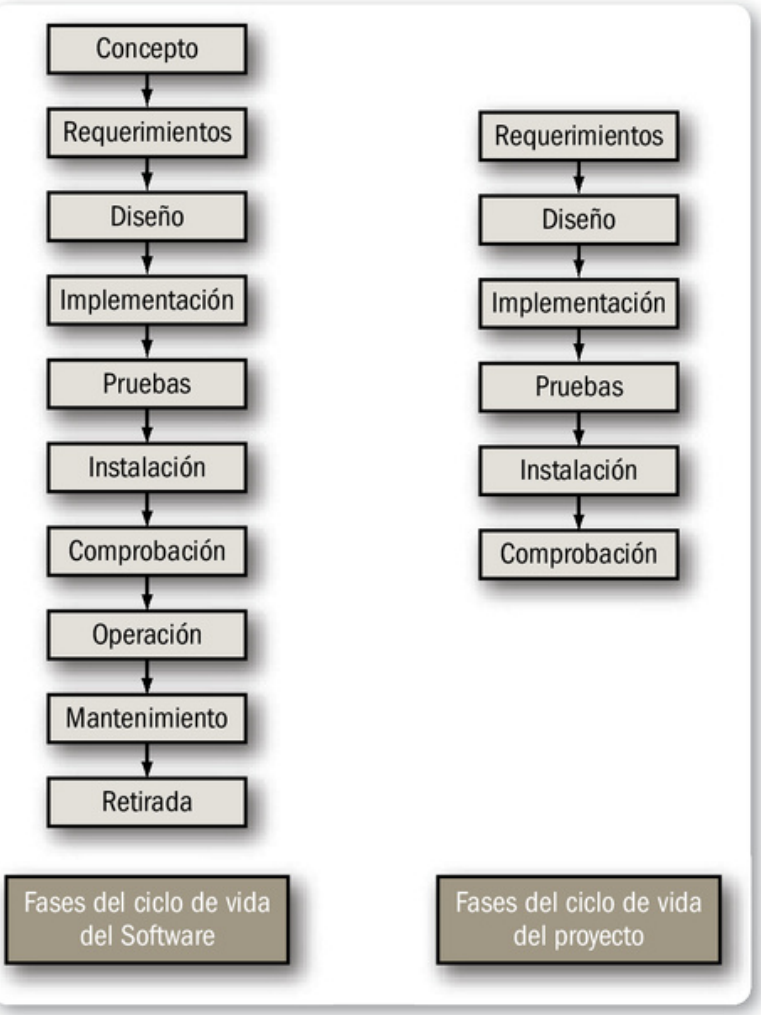


Figura 4. En estos esquemas podemos ver las etapas del ciclo de vida de un proyecto y de un software.

Modelos de ciclo de vida

Los modelos de ciclo de vida son aquellos que describen las fases principales del desarrollo del software y sus fases primarias esperadas. Son de gran utilidad para la administración del proceso y proveen de un espacio de trabajo para su definición.

Modelo en cascada

Es el enfoque metodológico que ordena rigurosamente las etapas del ciclo de vida del software, de modo que el inicio de cada etapa debe esperar a la finalización de la inmediatamente anterior.

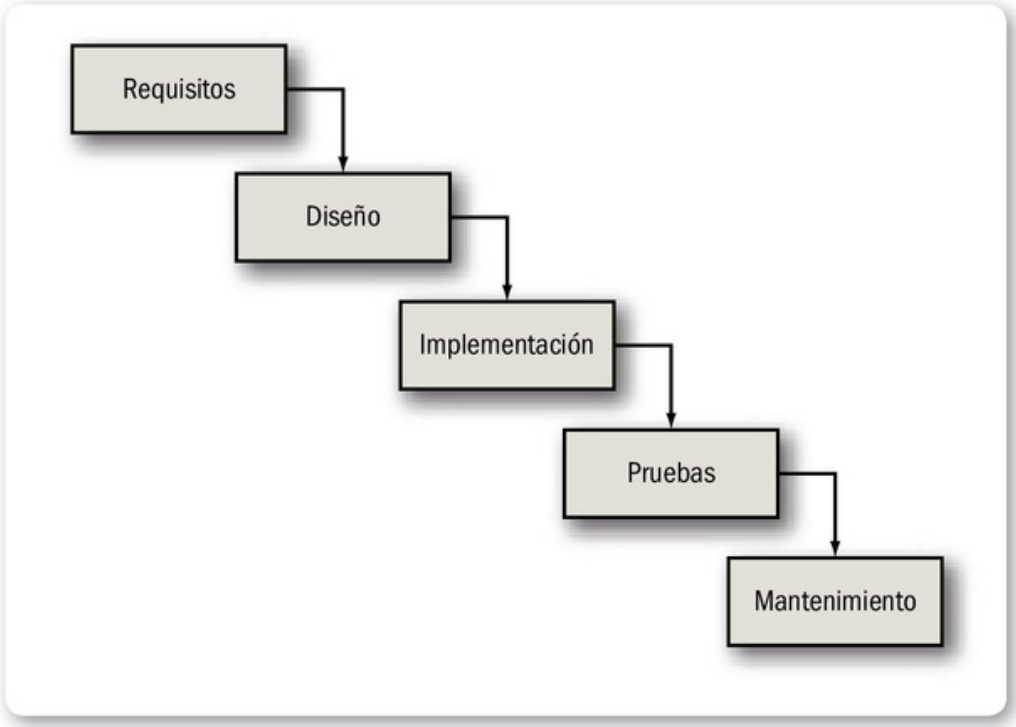


Figura 5. Modelo en cascada. Después de cada etapa, se realiza una revisión para comprobar si se puede pasar a la siguiente.

Modelo en V

Se desarrolló con el objeto de solucionar algunos problemas que ocasionaba el enfoque de cascada tradicional. En ese modelo, los defectos en el proceso se detectaban demasiado tarde en el ciclo de vida, ya que las pruebas no se introducían hasta el final del proyecto. Es por eso que el modelo en V sugiere que las pruebas comiencen a efectuarse en el ciclo de vida lo más pronto posible.

RUP (*RATIONAL UNIFIED PROCESS*)

Una de las metodologías pesadas más conocidas y utilizadas es la RUP (*Rational Unified Process*), que divide el desarrollo en cuatro fases que definen su ciclo de vida. Ellas son: inicio (su objetivo es determinar la visión del proyecto y definir lo que se desea realizar), elaboración (etapa en la que se determina la arquitectura óptima del proyecto), construcción (se obtiene la capacidad operacional inicial) y transmisión (permite obtener el producto acabado y definido).

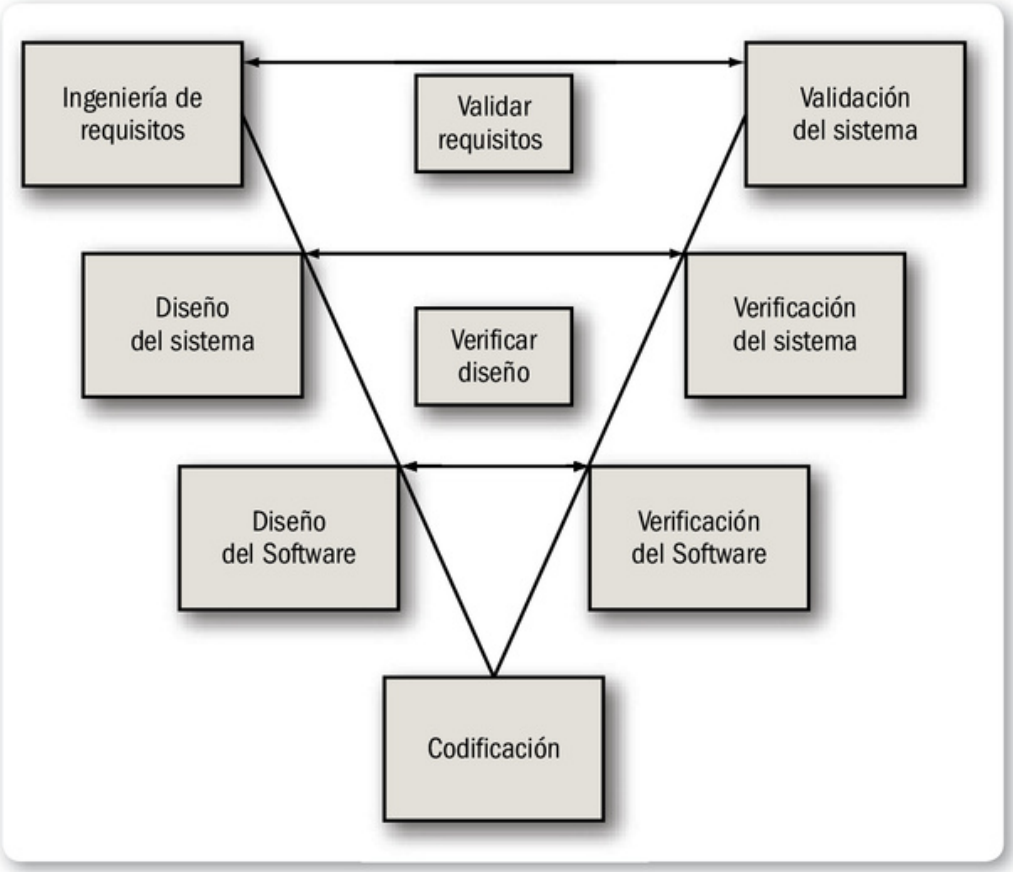


Figura 6. Modelo en V. Es aplicado por muchas compañías, debido a que se encuentra disponible públicamente.

Modelo iterativo

Es un modelo derivado del ciclo de vida en cascada, que busca reducir el riesgo que pueda surgir entre las necesidades del usuario y el producto final. Consiste en la iteración de varios ciclos de vida en cascada, en donde, al final de cada iteración, se le entrega al cliente una

UML

UML ofrece un estándar para describir un "plano" del sistema (modelo), incluyendo aspectos conceptuales tales como: los procesos de negocio, las funciones del sistema y los aspectos concretos. Estos últimos serían las expresiones del lenguaje de programación, los esquemas de bases de datos y los componentes reutilizables (librerías y clases). Web: **www.uml.org**

versión mejorada o con mayores funcionalidades del producto. El cliente es quien después de cada iteración evalúa el resultado y lo corrige o propone mejoras.

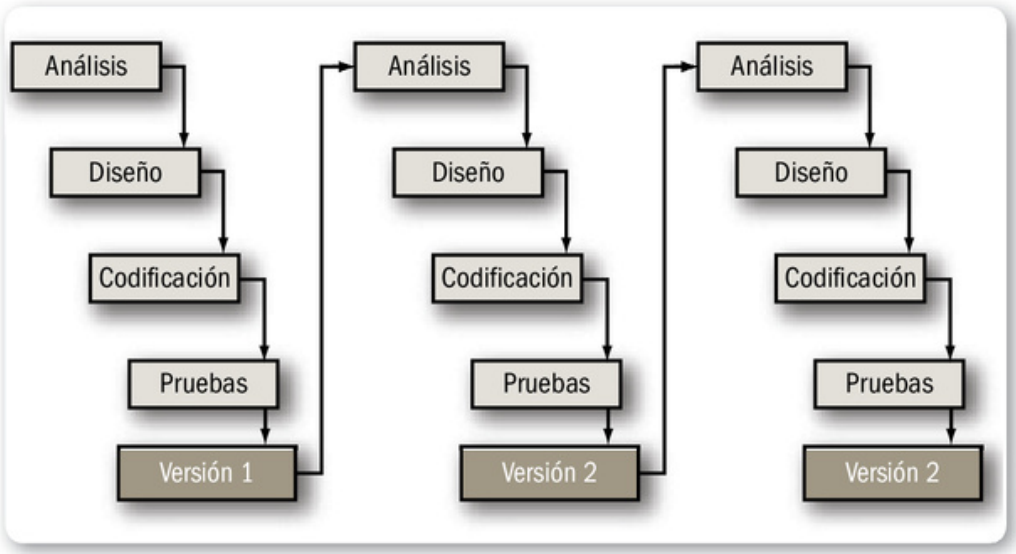


Figura 7. Modelo iterativo. Estas iteraciones se repiten hasta obtener un producto que satisfaga las necesidades del cliente.

Modelo de desarrollo incremental

El modelo incremental combina elementos del modelo en cascada con la filosofía interactiva de construcción de prototipos, basándose en la incrementación de las funcionalidades del programa. Aplica secuencias lineales de manera escalonada, mientras progresa el tiempo en el calendario, y cada secuencia lineal produce un incremento del software. El primer incremento es a menudo un producto esencial que reune solo los requisitos básicos, y se centra en la entrega de un producto operativo con cada incremento. Los

METODOLOGÍA VS. CICLO DE VIDA

El ciclo de vida indica qué es lo que hay que obtener a lo largo del desarrollo del proyecto, pero no menciona cómo hacerlo. Es la metodología la que indica cómo hay que obtener los distintos productos parciales y finales. Esta puede seguir uno o varios modelos de ciclo de vida.

primeros incrementos son versiones incompletas del producto final, pero proporcionan al usuario la funcionalidad que precisa y, también, una plataforma para la evaluación.

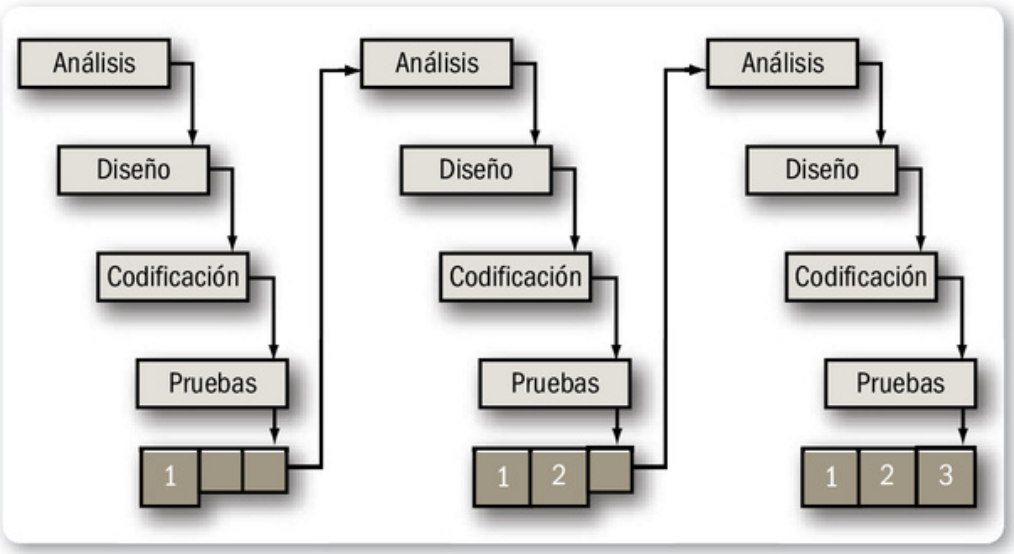


Figura 8. Modelo de desarrollo incremental. A partir de la evaluación, se planea el siguiente incremento, y así sucesivamente.

Modelo de prototipos

La construcción de prototipos comienza con la recolección de requisitos, y es en esa etapa cuando se reúnen desarrollador y cliente para definir los objetivos globales del software. Este modelo se centra en una representación de los aspectos del software, que serán visibles para el usuario/cliente y llevarán a la construcción de un **prototipo**. Este evalúa al cliente y refina los requisitos del software, de modo de permitir que el desarrollador comprenda mejor lo que se necesita hacer.

METODOLOGÍAS PESADAS

Como representantes de esta escuela, podemos nombrar a Winston Royce (Royce 1970) y Edward Yourdon (Yourdon 2009), entre otros. Estas metodologías suelen denominarse tradicionales y se inspiraron en otras disciplinas, tales como la ingeniería civil y la ingeniería mecánica. La más conocida de ellas es RUP (*Rational Unified Process*).

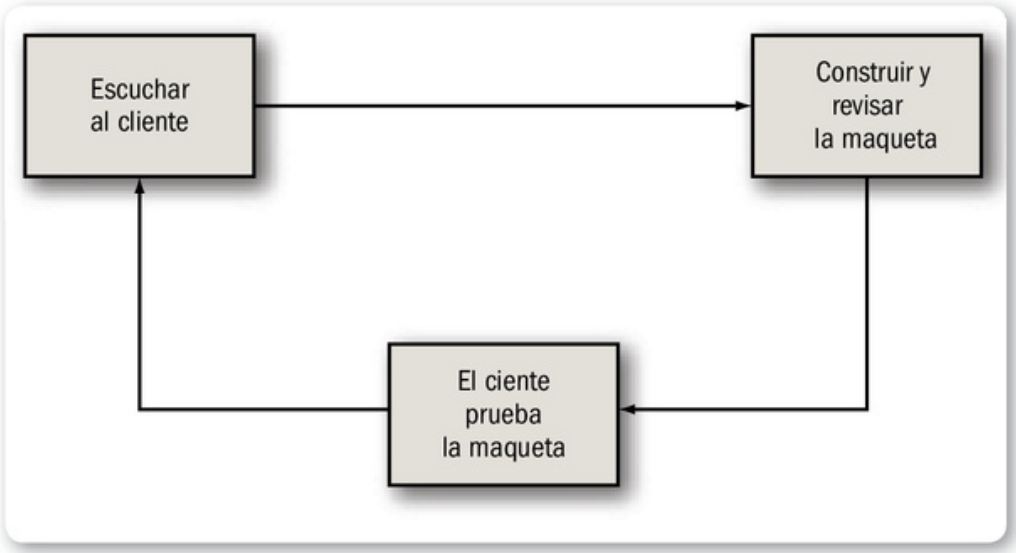


Figura 9. Modelo de prototipo. El diseño rápido se centra en representar los aspectos del software que serán visibles para el cliente.

Modelo en espiral

Este modelo combina las características del modelo de prototipos y el modelo en cascada, y fue creado para proyectos largos, complejos y de costo elevado. Un ejemplo puede ser la creación de un sistema operativo.

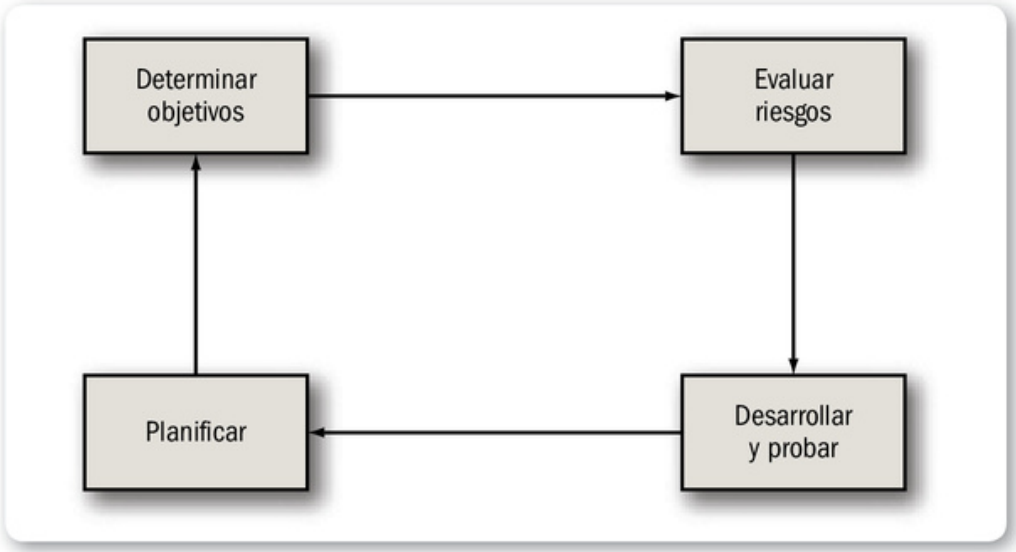


Figura 10. Modelo en espiral. Al terminar una iteración, se comprueba el cumplimiento de los requisitos establecidos y su funcionamiento.

Hasta aquí hemos visto los diferentes ciclos de vida que existen dentro del desarrollo de un software, considerados como proyectos que tienen un inicio y un fin. Si nos referimos a un equipo de personas que se dedicarán al desarrollo de aplicaciones, es importante tener en claro todos estos procedimientos. Si nuestro objetivo es ser desarrollador, debemos tener en cuenta que existirán distintos tipos de actores que realizarán alguna tarea en particular.

EXISTEN DISTINTOS TIPOS DE ACTORES QUE REALIZARÁN ALGUNA TAREA EN PARTICULAR

Todo lo que explicamos hasta el momento está orientado al desarrollo de aplicaciones con certificación de calidad, y nos será de gran utilidad a la hora de trabajar en equipo.

A continuación, veremos las generalidades que podemos encontrar dentro de las diferentes metodologías aplicables a nuestro proyecto.

Generalidades sobre metodologías

En esta sección vamos a conocer las diferentes metodologías que podemos aplicar a nuestro proyecto, qué son y para qué sirven, de modo de adquirir más fundamentos en nuestra elección final.

Desarrollo convencional (sin metodología)

- Los resultados finales son impredecibles.
- No hay forma de controlar lo que está sucediendo en el proyecto.
- Los cambios en la organización van a afectar en forma negativa al proceso de desarrollo.

Desarrollo estructurado

- Programación estructurada
- Diseño estructurado
- Análisis estructurado
- Especificaciones funcionales
- Gráficas
- Particionadas
- Mínimamente redundantes

Desarrollo orientado a objetos

Su esencia es la identificación y organización de conceptos del dominio de la aplicación, y no tanto su representación final en un lenguaje de programación.

- Se eliminan fronteras entre fases debido a la naturaleza iterativa del desarrollo orientado al objeto.
- Aparece una nueva forma de concebir los lenguajes de programación y su uso al incorporarse bibliotecas de clases y otros componentes reutilizables.
- Hay un alto grado de iteración y solapamiento, lo que lleva a una forma de trabajo muy dinámica.
- Son interactivas e incrementales.
- Es fácil dividir el sistema en varios subsistemas independientes.
- Se fomenta la reutilización de componentes.

Con todo lo que hemos analizaddo con respecto a los tipos de metodologías, ahora podemos seleccionar cuál es la más conveniente para implementar en nuestros desarrollos de software actuales y futuros. Para eso, es importante prestar atención a los siguientes capítulos, que nos permitirán seguir avanzando en el análisis del ciclo de vida.

Análisis de sistema

Anteriormente vimos los ciclos de vida y las metodologías que podemos emplear en algunos desarrollos de aplicaciones. Ahora nos centraremos en el análisis de los sistemas y las etapas fundamentales en dicho desarrollo. Cada paso es una forma ordenada y correcta de encarar un nuevo negocio para desarrollar, y a partir de allí, cada

METODOLOGÍAS ÁGILES

Las metodologías ágiles se caracterizan por estar más orientadas a las personas que al proceso, y por ser mucho más sencillas. Esto se debe a que son fáciles de aprender y se adaptan muy bien al medio, con lo cual permiten efectuar cambios de último momento. Una de las metodologías ágiles más utilizadas es SCRUMM.

empresa y programador deberán elegir alguno de estos caminos posibles. A continuación,veremos las etapas más comunes para el análisis de los sistemas, junto con un ejemplo práctico que nos ayudará a reconocerlas mejor.

Supongamos que una empresa vende espacios publicitarios que figuran en revistas y diarios. La organización siempre llevó el registro de los pedidos y presupuestos en forma manual, hasta que un día, decide implementar alguna aplicación informática para lograr que el proceso de ventas resulte más rápido y fiable.

> LUEGO DE ESTABLECER EL FUNCIONAMIENTO, DEBEMOS REALIZAR UN RELEVAMIENTO

Imaginemos que si por día quiere registrar a mano diez presupuestos de diferentes revistas, tendrá que recurrir a pilas de cajas llenas de papeles y archivos. Es por eso que será necesario incorporar alguna herramienta que permita mejorar las decisiones y el servicio a sus clientes, para cotejar información útil acerca de cuáles son los más activos y qué tipo de publicidades se venden mejor.

Estas tareas pueden variar en el análisis de un sistema. En primera instancia, tenemos el pedido del cliente, que determinamos como **requisito**. Hay una situación que él desea solucionar o mejorar, y que desembocará en la planificación de un proyecto. Al haber aclarado cuál es el funcionamiento o fin de la aplicación informática, debemos realizar, como analistas, un **relevamiento** de la información implicada. Cuando terminamos dicha tarea, podemos generar un **diagnóstico** de cómo encontramos los procesos actuales en los que se ve involucrada la organización con respecto a la futura herramienta informática.

Luego de recolectar la información, entramos en la etapa de **prototipado** o **diseño del sistema**, en la que volcamos dicha recolección de datos para dar un "rostro" a las aplicaciones. En esta

ETAPAS DE TESTING

Al igual que el desarrollo de software, las pruebas también tienen diferentes etapas que es importante tener en cuenta. Algunas de las más relevantes son: planificación y control, análisis y diseño, implementación y ejecución, evaluación y cierre.

fase estamos en condiciones de mostrarle a nuestro cliente una **propuesta** teórica acerca de cómo funcionaría el proyecto.

Una vez que terminamos de documentar y mostrar al cliente los avances de las etapas de relevamiento y prototipado, ya podemos entregar a los desarrolladores la estructura del software que ellos crearán por medio de un lenguaje de programación. Cuando ellos terminen de desarrollar el software, ya sea en forma parcial o completa, seguirán las etapas de **implementación** y **testing** del proyecto.

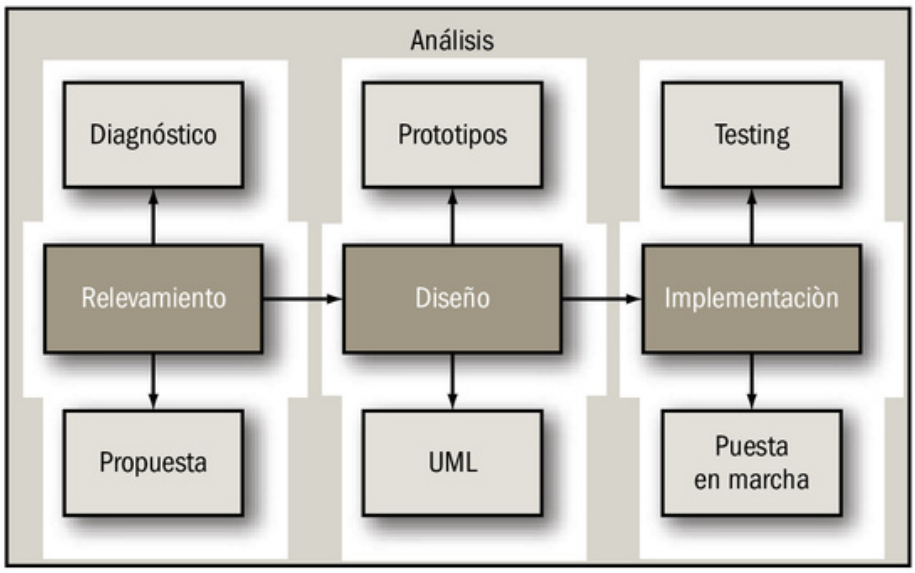


Figura 11. Las etapas del análisis se dirigen hacia las conclusiones de efectividad y eficiencia de los sistemas relevados.

El análisis de sistema estará involucrado en cada paso, ya que lleva a cabo la documentación y controla todas las tareas necesarias para que el proyecto funcione correctamente. A continuación, vamos a desarrollar en profundidad cada una de estas partes que constituyen el análisis.

Relevamiento

El proceso de relevamiento es fundamental en el diseño y la confección de un software, ya que nos permitirá comprender en detalle qué tareas están involucradas en el proceso que necesitamos solucionar

con nuestra aplicación informática. Para concretar este objetivo, vamos a revisar distintas técnicas de las que podemos valernos.

Cuando hacemos el relevamiento gracias a las visitas al cliente, debemos seguir algunos pasos, tales como: identificar las fuentes de información, realizar las preguntas apropiadas, analizar la información, confirmar con los usuarios y sintetizar los requisitos.

EXISTEN DISTINTAS TÉCNICAS PARA DETERMINAR QUÉ TAREAS ESTÁN INVOLUCRADAS

En la organización de nuestro ejemplo, los usuarios que más realizan este proceso de presupuesto y venta de espacios publicitarios son las personas que trabajan en Ventas y Atención al cliente. No obstante, también debemos tener en cuenta al personal que integra la Gerencia de la organización, ya que ellos son quienes llevan a cabo los controles.

Una vez que entramos en los procesos de negocio de un cliente u organización, veremos que no siempre encontraremos las puertas abiertas para recopilar información. Entonces, vamos a ver que existen varias técnicas que podemos utilizar.

Técnicas para recolectar información

- **Entrevistas**: recorremos la organización y logramos un contacto directo con los actores de los procesos por relevar. Dependiendo del tipo de preguntas que planteemos, vamos a obtener más o menos información valiosa. Es importante tener en cuenta a qué personas entrevistamos, porque si es alguien ajeno al proceso, puede perjudicar los requisitos iniciales con percepciones personales referidas al proceso.

ESTRUCTURADAS VS. NO ESTRUCTURADAS

Las entrevistas estructuradas se caracterizan por mantener un modelo rígido de preguntas que están planeadas de antemano, y no se permiten desviaciones. En cambio, las entrevistas no estructuradas son aquellas que pueden variar su plan original y admitir algunas variaciones em sus preguntas, en la medida en que el entrevistador lo considerere conveniente.

LAS ENTREVISTAS	
▼ VENTAJAS	▼ DESVENTAJAS
Es el medio más directo para obtener información.	Requieren más tiempo y dinero.
Las personas suelen ser más sinceras cuando hablan, que cuando escriben.	Las personas que serán entrevistadas deben ser elegidas cuidadosamente.
El entrevistador puede hacer preguntas abiertas y establecer una mejor relación con el entrevistado.	No se pueden utilizar para obtener información en gran cantidad.

Tabla 2. Aquí se ven las ventajas y desventajas principales de las entrevistas.

Antes de iniciar el proceso de la entrevista, deben establecerse ciertos puntos importantes. Primero, determinar qué información se desea obtener y quién entrevistará sobre la base de los objetivos planteados, para así planificar qué preguntas deberá hacer. Luego, hay que realizar una cita por anticipado con los entrevistados, para indicarles el objetivo de la tarea. Es importante elegir el lugar y el momento adecuados para la reunión, presentando el tema de la entrevista y explicando el proyecto sobre el cual se trabajará. Para finalizar, se resume la información recopilada, se revisa que no hayan quedado dudas y se aclaran los datos faltantes.

- **Observación** (directa o indirecta): la ventaja principal de la observación es que recopilaremos información directamente, tomando notas que describen las actividades y cómo estas se generan. En general, el propósito de la visita del analista debe darse a conocer a los miembros de la organización por medio de los mandos superiores. El analista no interrumpe a los trabajadores, pero cuando las personas están siendo observadas directamente, tienden a mejorar las funciones que llevan a cabo o, de lo contrario, molestarse por la presencia del observador. A veces es preciso efectuar varias visitas para generar confianza en presencia del analista.
  La observación directa nos lleva a participar en algunas actividades que observamos; en cambio, la indirecta implica observar como tercero o ajeno a los procesos, y solo relevar la información.

- **Estudio de documentación**: una de las tareas principales que debemos realizar, es revisar aquellos registros que se efectúan en la organización. Más allá de que se trate de escritos, fotocopias, documentos o digitales, tendremos que analizar la información para diagnosticar los procesos que se llevan a cabo. A partir de allí, se evaluará cuál es la mejor manera de manejar dicha documentación o proponer un cambio de procedimientos, en caso de ser necesario.
- **Cuestionarios**: herramienta útil, basada en una serie de preguntas escritas a las que hay que contestar también por escrito.

CUESTIONARIOS

▼ VENTAJAS	▼ DESVENTAJAS
Se obtiene un alto volumen de información a un costo bajo y en menor tiempo.	Tienen limitaciones sobre el tipo de preguntas que se pueden realizar.
Eliminan cualquier influencia sobre quien contesta.	Suelen ocurrir problemas de interpretación.
La información es sincera, ya que puede ser anónima.	Si no hay control, pueden tener una tasa de retorno muy baja y una muestra pequeña.

Tabla 3. Ventajas y desventajas de los cuestionarios.

- **Tormenta de ideas (brainstorming)**: es una técnica muy utilizada en distintos ámbitos profesionales, por ejemplo, en publicidad. Su trabajo grupal facilita el surgimiento de nuevas ideas sobre un tema o problema determinado, en un ambiente relajado. De esta manera, podemos vincularla a los procesos y a las mejoras de utilidad que se logran con un sistema informativo.

Es posible utilizar varias herramientas para recopilar información, pero ¿cuál es la más efectiva? Eso dependerá del tipo de organización y del proceso que necesitemos relevar. En el caso de nuestro ejemplo, la aplicación es concreta y pequeña; entonces, debemos utilizar herramientas puntuales, como revisar la documentación y las entrevistas de los usuarios que llevan a cabo dichas tareas.

Para seguir avanzando, vamos a desarrollar el diseño de un sistema. En este caso, será fundamental contar con el proceso del relevamiento.

Diseño de un sistema

A la hora de diseñar la aplicación de software, vamos a dividir el proceso en dos partes: el diseño de prototipos (las ventanas que involucra nuestro desarrollo) y el diseño del funcionamiento (el mecanismo interno de las operaciones de desarrollo). Para efectuar esta tarea, podemos utilizar herramientas de modelado que nos permitirán crear una representación que refleje aspectos de nuestro sistema. Por ejemplo, podemos trabajar con **UML (lenguaje de modelado unificado)**, que nos facilitará el diseño del sistema, al permitirnos usar varios tipos de herramientas.

Figura 12. UML se encuentra respaldado por OMG (*Object Management Group*).

Recomendamos visitar la Web y bibliografía sobre UML, ya que sus herramientas son variadas. En nuestro caso, repasaremos algunas de ellas, que nos serán de utilidad. Dentro de los programas que pueden ser prácticos para el modelado, se encuentran:

UML, UNIFIED MODELING LANGUAGE

Es el lenguaje de modelado de sistemas de software más conocido y utilizado. Es un lenguaje gráfico para visualizar, especificar, construir y documentar un sistema. UML ofrece un estándar para describir un "plano" del sistema (modelo). Web: **www.uml.org**.

UML Tutoriales y Software:

- **https://secure.nomagic.com**: No Magic - MagicDraw UML
- **www.borland.com**: Borland's UML TutorialCetus Links - UML Tutorials
- **www.jeckle.de**: Mario Jeckle - UML Tutorials
- **www.sparxsystems.com**: Sparx Systems' UML 2.0 Tutorial

Diagrama de casos de uso

Los diagramas de casos de uso son los más empleados en los proyectos de desarrollo de sistemas informáticos. A continuación, veremos en detalle cuáles son sus componentes y, luego, los aplicaremos a un ejemplo concreto.

EL DIAGRAMA NOS MUESTRA EL FUNCIONAMIENTO Y LOS ACTORES INVOLUCRADOS

Un diagrama de casos de uso es un esquema de comportamiento que define por medios gráficos las representaciones de casos de negocio u operaciones de una situación determinada. Por ejemplo: podemos utilizarlo para registrar las ventas, realizar un inventario de productos, registrarse en una web, etc.

Los casos de uso sirven para darle al cliente una vista general y simple de un proceso de negocio, ya que suelen estar dirigidos a personas que no tienen conocimientos sobre programación. De esta forma, podemos explicarles el funcionamiento del sistema y los actores involucrados en la interacción con él.

Para comprender mejor cómo están compuestos los casos de uso, es importante conocer sus diferentes componentes. En todo caso de uso siempre hay un **actor** que inicia y **otro actor** (puede ser el mismo de antes o no) que recibe algo por parte del **sistema**. A continuación, veremos las herramientas que nos ofrece.

OPEN SOURCE

Open Source, o código abierto, es el término que se utiliza para los programas distribuidos y desarrollados libremente. El código abierto tiene un punto de vista orientado a los beneficios prácticos de compartir el código a quien lo desee. Es diferente de las normas que tiene en cuenta el software libre.

Actores

Los casos de uso están típicamente relacionados con los actores, que son entidades humanas o máquinas que interactúan con el sistema para llevar a cabo un trabajo significativo que ayude a alcanzar una meta. El conjunto de casos de uso de un actor define su rol global en el sistema y el alcance de su acción.

Figura 13. Aquí la herramienta UML es actor.

Caso de uso (elipse)

En la elipse ubicamos una funcionalidad o servicio provisto por el sistema, que va a interactuar con los actores u otros servicios del sistema. Por lo general, escribimos algo breve que haga referencia a una actividad, como Registrar presupuesto.

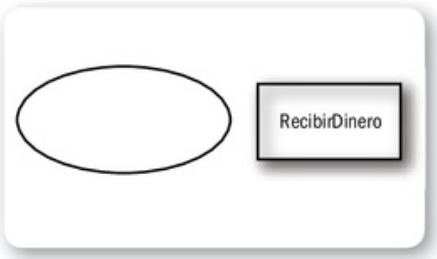


Figura 14. Aquí la herramienta UML es caso de uso.

Límite del sistema (escenario)

En un recuadro se encierran los casos de uso, y este representa el límite del sistema. Solo contiene comportamientos generales de importancia, siendo estos los que utilizan los actores del sistema.

Relaciones

Los casos de uso pueden tener relaciones con otros casos de uso. Los tres tipos de relaciones más comunes entre ellos son:

- **<<include>> / <<incluir>>**: especifica una situación en la que un caso de uso tiene lugar dentro de otro caso de uso.
- **<<extends>> / <<extender>>**: especifica que, en ciertas situaciones o en algún punto (llamado punto de extensión), un caso de uso será extendido por otro.
- **Generalización o herencia**: un caso de uso hereda las características del «súper» caso de uso, y puede volver a especificar algunas o todas de una forma similar a las herencias entre clases.

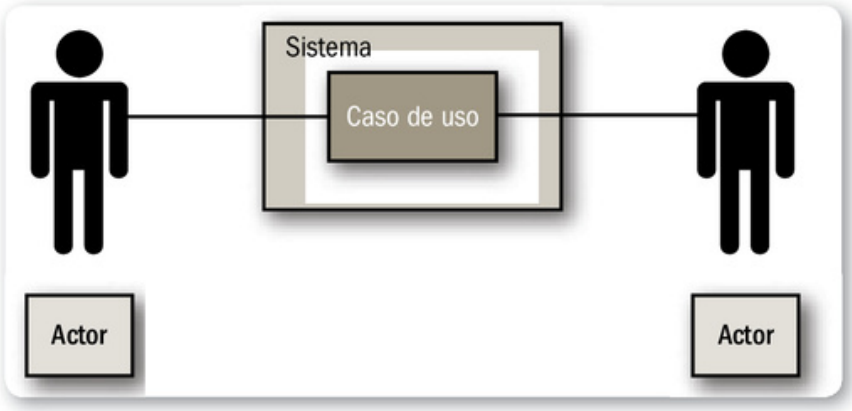


Figura 15. En este esquema podemos ver el modelo de un caso de uso, representado por la elipse.

En la **Figura 15**, las dos figuras en los extremos izquierdo y derecho representan a los actores que intervienen. El actor que inicia se encuentra a la izquierda del caso de uso, y el que recibe, a la derecha.

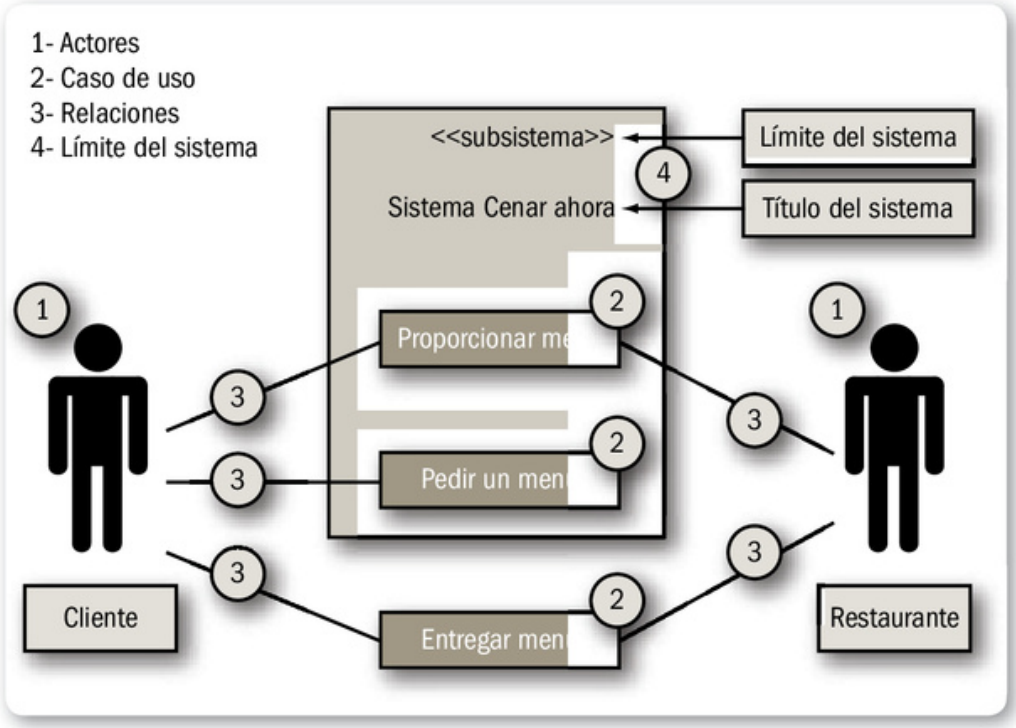


Figura 16. Ejemplo de un modelo de un caso de uso, donde vemos involucradas la mayoría de las herramientas y su forma de declararlas.

El nombre del actor aparece justo debajo de él, y el del caso de uso aparece dentro de la elipse o justo debajo de ella. Una línea asociativa conecta a un actor con el caso de uso, y representa la comunicación entre ellos. La línea asociativa es sólida, y el rectángulo envuelve a los casos de uso dentro del sistema.

Los casos de uso son una gran herramienta para representar modelos de negocios que relevamos antes. Así, podremos mostrarles a nuestros clientes, por un medio gráfico, cómo funcionaría el sistema y quiénes estarían involucrados; y a los desarrolladores, cómo debería funcionar la aplicación según la visión de los clientes.

Sobre este modelado hay mucho material gratuito para consultar en la Web. En las siguientes imágenes mostramos algunos ejemplos que podemos encontrar.

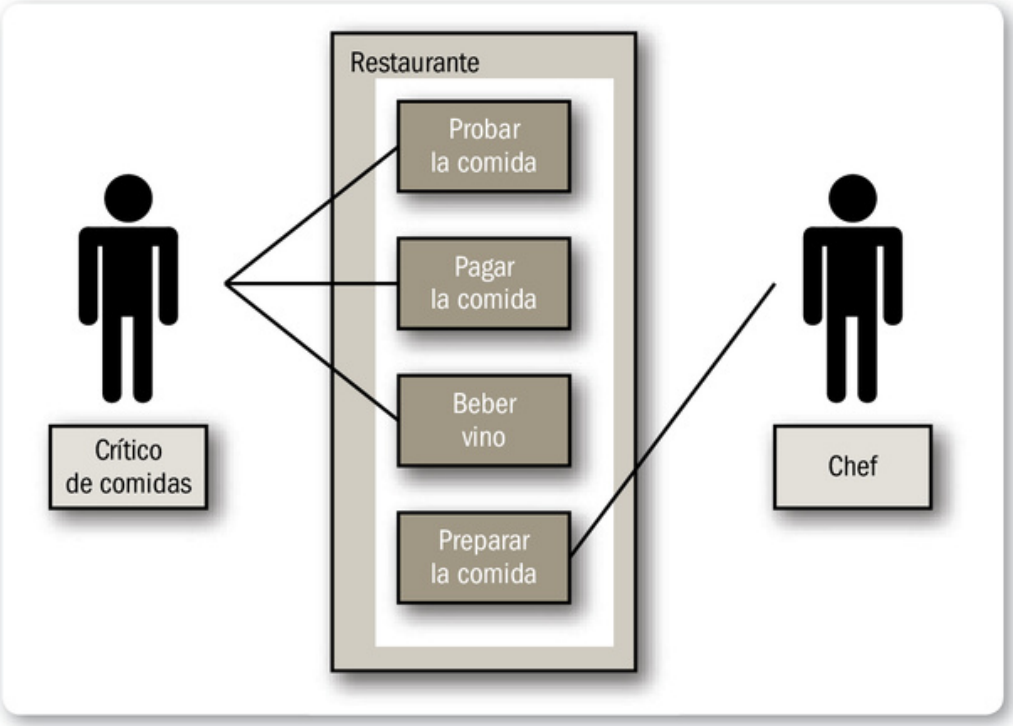


Figura 17. En este restaurante tenemos dos actores y cuatro globos de procesos que se ven afectados por acciones sobre un sistema.

Continuando con nuestro ejemplo de presupuesto y ventas de publicaciones en revistas, después del relevamiento realizado, podemos definir el siguiente caso de uso.

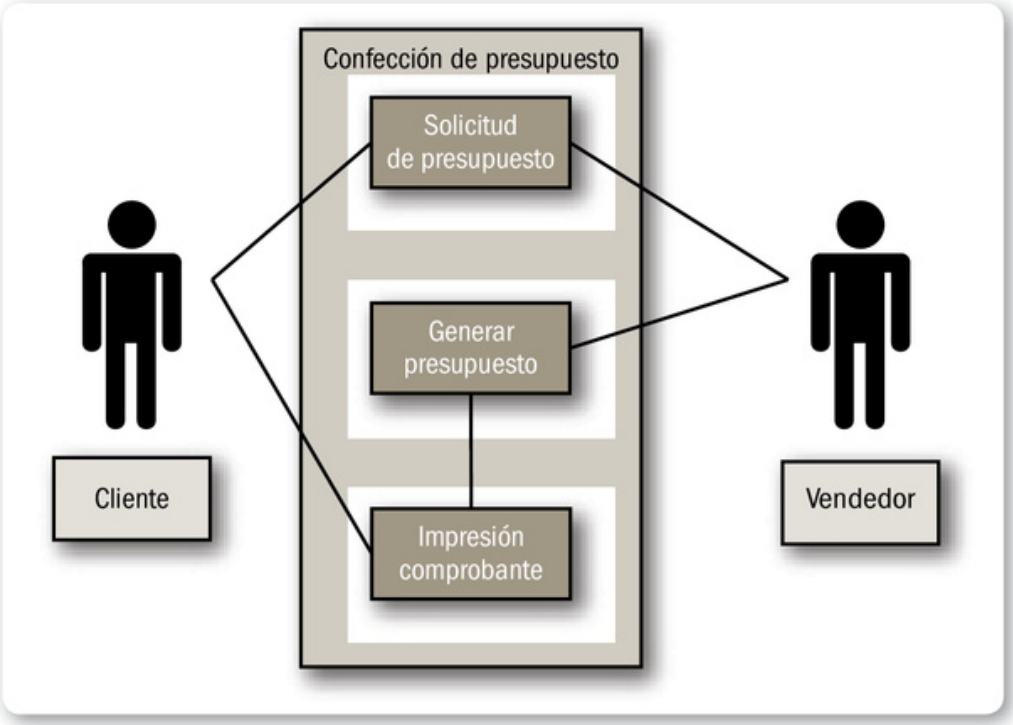


Figura 18. Ejemplo de Presupuestos. En este caso, los actores se ven involucrados en un mismo proceso, donde interactúan.

Prototipos

Anteriormente vimos modelos que grafican los procesos de nuestro desarrollo del sistema, y que sirven para que el cliente conozca, en teoría, cómo funcionará la aplicación que vamos a confeccionar; luego, lo entregaremos a los desarrolladores. En esta sección veremos que, además de estos modelos, también podemos confeccionar prototipos de interfaces gráficas que nos demuestren cómo se verán nuestras aplicaciones luego de ser desarrolladas.

Los prototipos son un ejemplo o molde en el que "fabricaremos" una figura u otra. Dentro del desarrollo del software, sería la interfaz gráfica o modelo de funcionamiento gráfico que permite mostrarle a un usuario cómo será el aspecto o el funcionamiento del futuro desarrollo.

El uso de prototipos es una poderosa herramienta que nos facilitará la comunicación con el usuario de las aplicaciones, junto con sus reacciones y apreciaciones. De esta forma, veremos que este actúa como maqueta, y no precisamente como producto final.

Podemos conceptualizar dos grupos principales de prototipos: un **prototipo desechable**, que sirve como una vasta demostración de los requisitos, que luego se desecha y hace un paradigma diferente; y un **prototipo evolutivo**, que emplea el prototipo como primera evaluación del sistema terminado.

Objetivo de los prototipos

Al presentar prototipos, estamos muy interesados en las reacciones de los usuarios y en los comentarios sobre cómo sería el manejo desde su punto de vista. Aquí vamos a ver en detalle la manera en que reaccionan, y cómo es el ajuste entre sus necesidades y las características del producto.

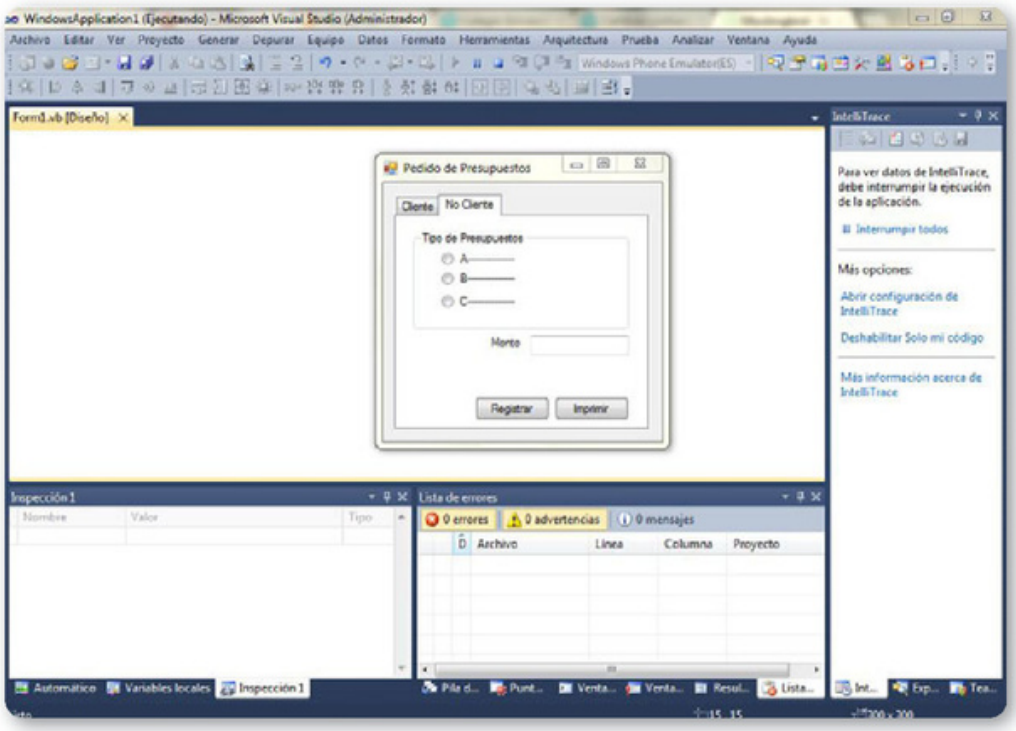


Figura 19. En esta pantalla podemos ver un claro ejemplo de prototipos para presupuesto.

Las reacciones deben recopilarse con las herramientas utilizadas para el relevamiento y diseñadas para recoger la opinión de cada persona sobre el prototipo analizado. Evaluando estas perspectivas, podemos

llegar a percibir si a los usuarios les agrada o no el sistema, e incluso evaluar si habrá dificultades para su posterior venta o implementación.

ANALIZANDO LAS PERSPECTIVAS PODEMOS EVALUAR SI EL SISTEMA ES EL ADECUADO

En general, para evaluar si una aplicación es adecuada para la creación de prototipos, basta asegurarse de que nos permita crear pantallas visuales dinámicas, interactuar intensamente con la persona, y demandar algoritmos o procesamiento de combinaciones en modo progresivo. Es preciso tener en cuenta que un prototipo puede ser desechado; por lo tanto, cualquier lenguaje de programación que nos permita graficarlo será suficiente.

Podemos ampliar más sobre este importantisimo tema utilizado en el desarrollo de aplicaciones consultando otro libro de nuestra editorial: *UML*, por Fernando Asteasuain.

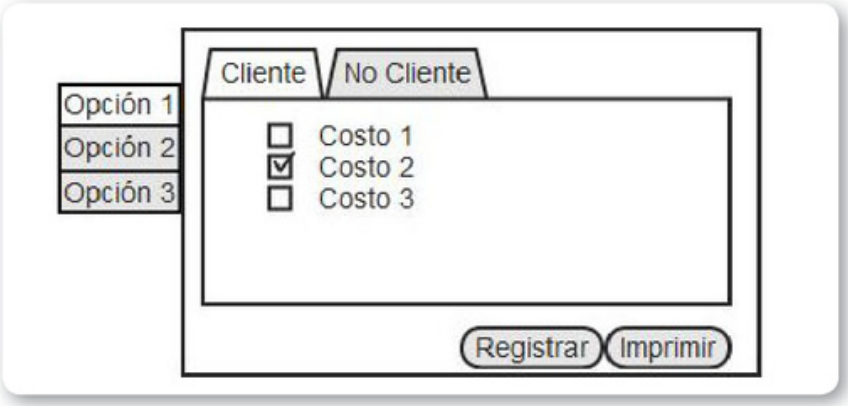


Figura 20. Ejemplo de prototipos con mockingbird.

CONFECCIÓN DE PROTOTIPOS

Podemos utilizar las facilidades que nos brindan algunos IDE (entorno de desarrollo integrado), por ejemplo, el de Visual Studio, para crear formularios con controles. También existen otras aplicaciones de maquetación que pueden sernos de utilidad, como: **https://gomockingbird.com**,**http://balsamiq.com**,**http://pencil.evolus.vn** y**Microsoft Visio**.

Implementación del desarrollo

El momento de insertar el software, ya sea en el negocio de nuestro cliente o en el lugar de aplicación que hayamos elegido, es una etapa crucial para nuestro trabajo. En caso de ser un software a pedido de un cliente, tenemos que asegurarnos de que se hayan aprobado todos los puntos anteriores de análisis. Es decir, que la información haya sido correctamente recopilada, que el cliente haya comprendido cómo funcionará el sistema y que se hayan aprobado los prototipos del sistema. Luego de haber desarrollado el sistema, entramos en la etapa de implementación. Tengamos presente que todas estas instancias deben llevarse de una manera organizada y comprometida, ya que de esto dependerá el correcto análisis y desarrollo del software.

> EL MOMENTO DE INSERTAR EL SOFTWARE ES UNA ETAPA CRUCIAL PARA NUESTRO TRABAJO

A continuación, veremos qué aspectos debemos tener en cuenta al momento de instalar el sistema.

Prueba o testing de aplicaciones

Este proceso de testing implica someter un software a ciertas condiciones para demostrar si es válido o no. De esta forma, podemos verificar si se ajusta a los requerimientos y validar que las funciones se implementen correctamente. Al considerar y analizar los resultados generados, se agrega valor no solo al producto, sino también a todo el proceso de desarrollo. Los valores de calidad que tienen mayor relevancia en las aplicaciones son: usabilidad, funcionabilidad, fiabilidad, seguridad, eficiencia y mantenimiento.

Testear y documentar una aplicación es un paso fundamental para asegurar la calidad del producto. Para hacerlo, existen herramientas

de software para proyectos web, como las que vimos anteriormente. La tarea de testing no es menor, y es necesario tener un gran cuidado al momento de la implementación, ya que es una misión específica de un equipo o integrante del desarrollo de software. La ventaja de esos programas de testing es que, en su mayoría,son **open source**.

Capacitación y formación del usuario

Para darle cierre al análisis de sistema, llega una de las tareas particularmente humanitarias: capacitar al usuario para el uso del sistema informático. En esta etapa es importante tener en cuenta que no todos los futuros usuarios se desenvuelven fácilmente en el manejo de los equipos y el software. En este punto, además de armarnos de paciencia para realizar la capacitación, debemos establecer claramente los objetivos de la tarea, buscar métodos variados (teóricos, prácticos o teórico-prácticos), escoger un lugar adecuado (espacio relajado que no distraiga a las personas) y utilizar materiales que sean comprensibles.

> ES IMPORTANTE QUE LA CAPACITACIÓN SE ENCARE DESDE EL PUNTO DE VISTA DEL USUARIO

La capacitación debe encararse desde el punto de vista de los usuarios. A veces puede ser útil preguntarles, durante la capacitación, cómo hacían antes ciertos procesos o tareas. De esta forma, logramos involucrarlos en la formación y mostrarles todas las ventajas que les ofrece el software en sus procesos cotidianos.

Dependiendo del tipo de actividad que se vaya a desempeñar, la capacitación puede acompañarse con algunos manuales de usuario; por ejemplo, si se trata de tareas regulares o específicas de una vez al mes, como es el caso del cierre de ventas o el reconteo del inventario.

Cerrando con este capítulo de análisis de sistema, podemos ver lo amplio que es un equipo de desarrollo y las variadas tareas que lo conforman. También vimos que cada una de ellas no es independiente del resto, sino que, de algún modo, siempre están relacionadas.

Para completar las metodologías y el análisis de sistema, contamos con el gráfico de la **Figura 21**, que nos permite ver la vinculación e inclusión de cada una de las partes.

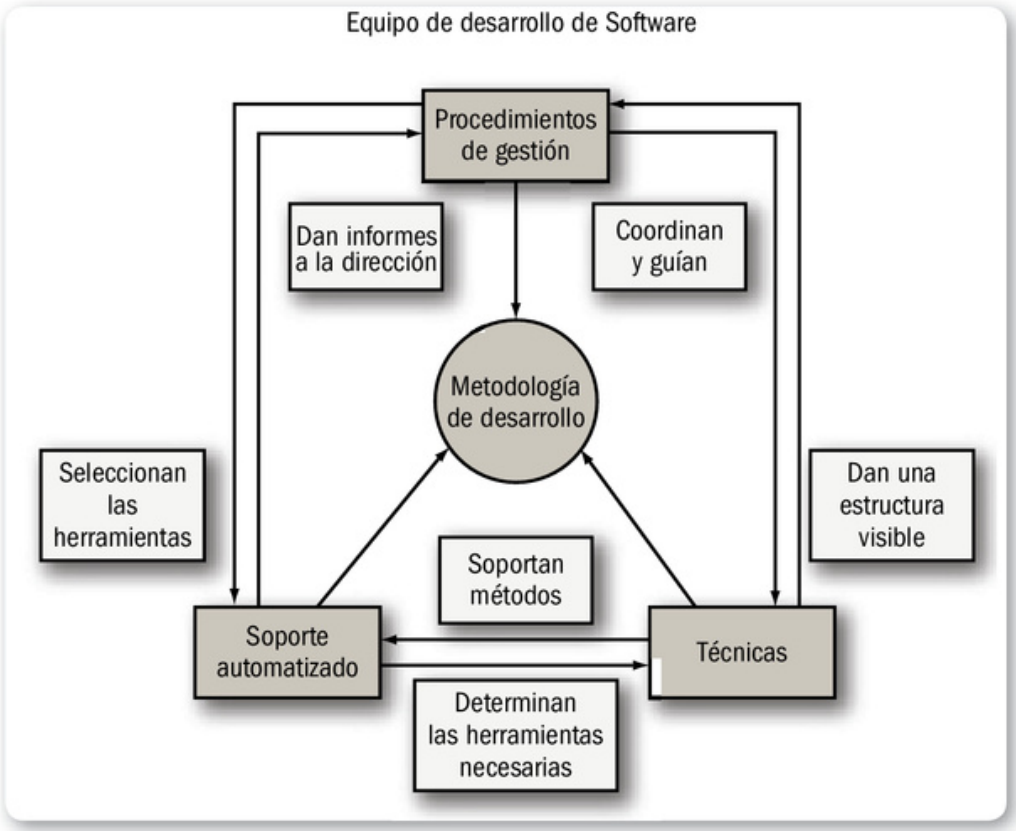


Figura 21. En la integración de metodologías y equipos de trabajo vemos los diferentes elementos que entran en juego y sus relaciones.

RESUMEN

Para comprender que los desarrollos de sistemas o aplicaciones son más amplios que simples programas de registros, hemos visto en este capítulo todas las etapas que constituyen su análisis. El proceso comienza con la necesidad o requerimiento del cliente; luego sigue el relevamiento minucioso de la información, las propuestas acerca de cómo funcionaría el sistema, el desarrollo específico, la puesta a prueba y, finalmente, la capacitación de los usuarios. Todo esto, gracias a una metodología de trabajo que vamos a utilizar. No olvidemos que, en cualquiera de estas etapas, es posible volver atrás, ya sea para corrección de errores o debido a los requerimientos de cada situación en particular.

Actividades

TEST DE AUTOEVALUACIÓN

1. Clasifique los tipos de metodología de desarrollo de software existentes, indicando brevemente sus características principales.
2. ¿Qué metodología utilizaría para realizar una aplicación que controle el funcionamiento de un conjunto de robots automatizados en una planta de ensamble de autos?
3. ¿Qué metodología emplearía en caso de realizar una aplicación que consista en la gestión y mantenimiento de una gran base de datos?
4. Describa los pasos generales que puede realizar en el análisis de sistema.
5. ¿Qué rol cumple el desarrollador de software en el análisis de sistema?
6. ¿Qué es UML?
7. ¿Qué es el prototipado?
8. ¿Qué métodos puede utilizar para mostrar a su cliente ejemplos de cómo funcionaría el sistema?
9. ¿Es correcto decir que solamente con saber qué hay que desarrollar podemos generar cualquier sistema que nos propongan? ¿Por qué?
10. ¿Es necesario interiorizarse en los procesos de un negocio para realizar un buen desarrollo?

*03 –

Ingreso al mundo de la programación

En este capítulo veremos por qué maneras podemos optar para transmitir a los equipos informáticos las indicaciones de tareas específicas que deseamos realizar. Para lograr que la interpretación de estas indicaciones sea correcta, vamos a analizar los medios necesarios.

- **La lógica de un humano y de una máquina 70**
- **Pseudocódigo: el lenguaje humano 71**
  - Qué son y cómo se usan las variables 75
  - Cómo se utilizan los operadores 80
- **Todo tiene un orden en la programación 94**
- **Tipos de datos estructurados 109**
  - Vector 110
  - Matriz 114
- **Utilizar funciones y procedimientos 120**
- **Resumen 129**
- **Actividades 130**

La lógica de un humano y de una máquina

El pensamiento lógico en los humanos es interpretado como el orden que este debe tener, indicando las operaciones de entendimiento en su movimiento hacia un objetivo. Anteriormente vimos que el algoritmo es un conjunto finito ordenado de pasos que nos lleva a la solución de un problema u objetivo. A lo largo de este capítulo, entenderemos que esto no difiere mucho del proceso lógico de una computadora.

> EL PENSAMIENTO LÓGICO DE LAS COMPUTADORAS SE BASA EN UN LENGUAJE BINARIO

La historia de la lógica para la computación comienza con la Revolución Digital, que se inició con la invención de la computadora digital y el acceso universal a las redes. **Alan Turing** fue quien unió la lógica y la computación, antes de que cualquier computadora fuese inventada.

El fue matemático y lógico, pionero en la teoría de la computación, y contribuyó con importantes análisis lógicos sobre los procesos computacionales. Las especificaciones para la computadora abstracta que él ideó, llamada la máquina de Turing, resultó ser una de sus contribuciones más relevantes a la teoría de la computación. Además, probó la posibilidad de construir una máquina universal que hiciera el trabajo de cualquiera diseñada para resolver problemas específicos, gracias a una programación adecuada. La máquina propuesta por Turing es un dispositivo relativamente simple, pero capaz de efectuar cualquier operación matemática. De esta forma, sería capaz de hacer todo aquello que fuera posible para el **cerebro humano**, incluyendo la capacidad de tener conciencia de sí mismo.

Pese a ser considerados formalmente equivalentes, los distintos modelos de computación presentan estructuras y comportamientos internos diferentes. Si bien el pensamiento lógico de las computadoras está basado en la lógica del humano, la forma de procesar esta lógica se basa en un lenguaje binario. Frente a esto, la pregunta sería: ¿qué lenguaje podemos utilizar como humanos, para que, luego, las máquinas interpreten las tareas que les queremos indicar? Es aquí donde entra en juego el pseudocódigo.

Pseudocódigo: el lenguaje humano

Debido a que no podemos programar rápidamente en lenguaje máquina (código binario), necesitamos adaptar de alguna manera el lenguaje humano a formas lógicas que se acerquen a las tareas que puede realizar una computadora. En programación, el lenguaje artificial e informal, pseudocódigo, es útil para desarrolladores en la confección de algoritmos, pero este no es un lenguaje de programación. El pseudocódigo describe algoritmos que podemos utilizar como una mezcla del lenguaje común (protocolo humano) con instrucciones de programación. Su objetivo principal es que el desarrollador se centre en la solución lógica y, luego, tenga prioridad en la sintaxis de un lenguaje de programación por utilizar.

En esencia, el pseudocódigo se puede definir como un lenguaje de especificaciones de algoritmos que indica, en palabras, los pasos que debe seguir un algoritmo para dar solución a un problema determinado. A continuación, explicaremos las normas más importantes que hay que tener en cuenta para desarrollarlo.

Normas para el pseudocódigo

Como ya mencionamos, el pseudocódigo es parecido a un lenguaje de programación en su escritura y, como tal, contiene un determinado **léxico**. Se trata de letras o caracteres que serán válidos para escribir las instrucciones que deseamos transmitir. La **sintaxis** es la especificación de palabras clave en combinación con otras que usaremos para formar las oraciones. Por último, la **semántica** es el significado que les daremos a dichas frases.

LÓGICA MATEMÁTICA

Es la disciplina que se vale de métodos de análisis y razonamiento, utilizando el lenguaje de las matemáticas como un lenguaje analítico. La lógica matemática nos ayuda a establecer criterios de verdad, y su importancia en la actualidad se debe al destacado papel que tiene en los diversos campos de la Informática.

Como hemos visto en capítulos anteriores, existen dos modos de representar algoritmos: gráficos con diagramas de flujo o sintaxis como pseudocódigo. Las ventajas de utilizar un pseudocódigo, en vez de un diagrama de flujo, es que ocupa menos espacio en la hoja de papel, permite representar fácilmente operaciones repetitivas complejas, simplifica el pasaje de un pseudocódigo a un lenguaje de programación y permite observar con claridad los niveles que tiene cada operación.

Una de las normas generales que encontraremos en la mayoría de los pseudocódigos y codificación en lenguaje de programación es la estructura secuencial. Su definición es una acción o instrucción que sigue a otra en secuencia. Las tareas se suceden de tal modo que la salida de una es la entrada de la siguiente, y así hasta el fin del proceso. A continuación, en la **Figura 1**, vemos cómo se representa una estructura secuencial en pseudocódigo.

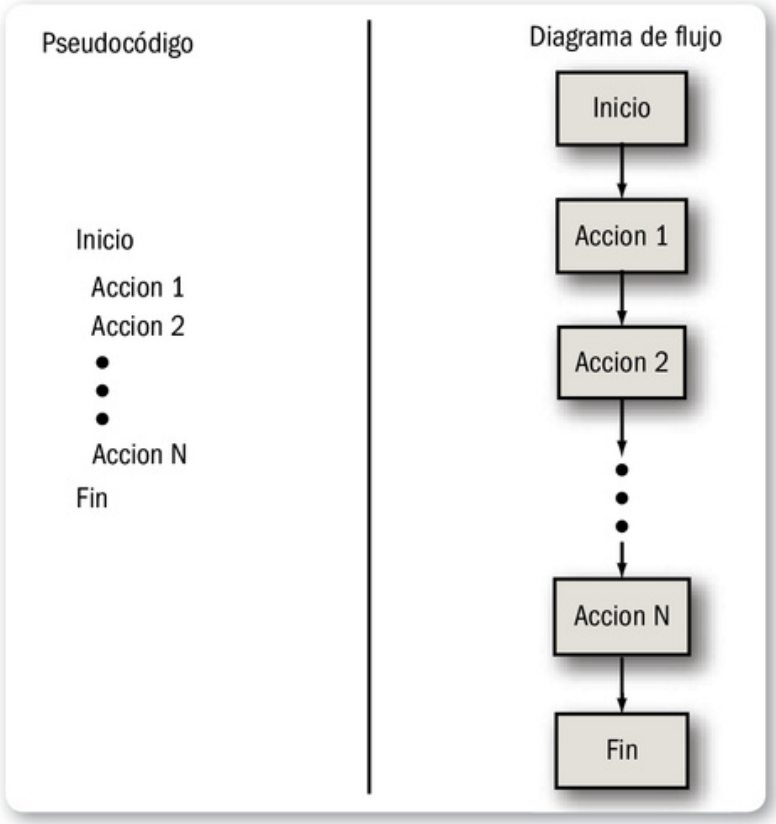


Figura 1. Comparación de pseudocódigo y diagrama de flujo que nos ayuda a comprender el funcionamiento del código por desarrollar.

Para comprender más sobre el uso de pseudocódigo y la estructura secuencial, a continuación realizaremos un caso práctico. En este ejemplo, lo representaremos con la puesta en marcha de un automóvil.

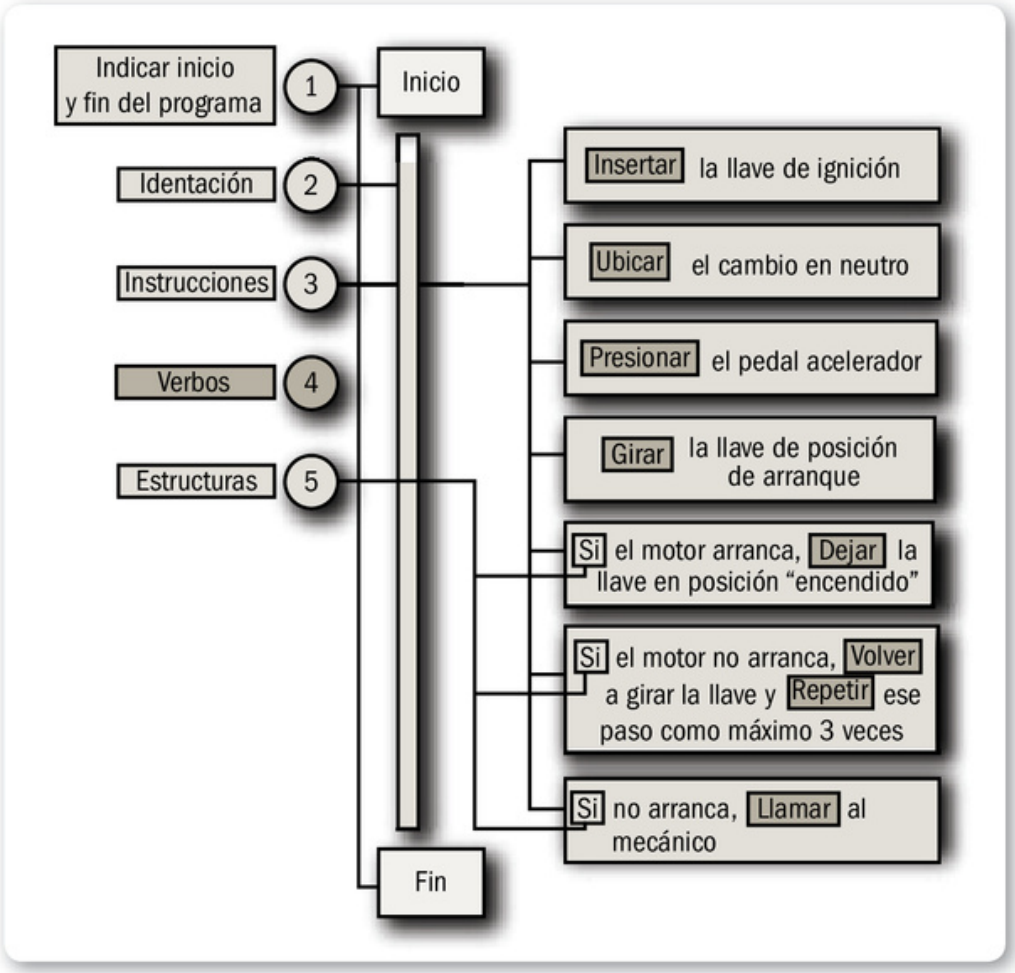


Figura 2. En esta figura podemos observar la explicación de pseudocódigo para arrancar un automóvil.

NASSI-SCHNEIDERMAN

El diagrama estructurado N-S es una técnica híbrida entre Diagramas de Flujo y Pseudocódigo. Esta técnica, también conocida como Diagrama de Chapín, utiliza una serie de cajas, similar a los diagramas de flujos, pero no requiere la utilización de flechas, debido a que su flujo siempre es descendente. Las acciones sucesivas se pueden escribir en cajas sucesivas y es posible escribir diferentes acciones.

Para resumir algunas de las normas generales que vemos en el ejemplo, podemos decir que: la estructura es secuencial, se indica el INICIO y FIN del programa, en el margen izquierdo se deja un espacio llamado indentación para identificar fácilmente las estructuras, y cada instrucción comienza con un verbo.

Tipos de datos

Para representar la información o las reglas que permitan cambiar fórmulas matemáticas a expresiones válidas de computación, es necesario tener en cuenta los tipos de datos. Las cosas se definen en la computadora mediante datos y algoritmos que van a operar sobre esos datos. A nivel de la máquina, estos datos se representan como una serie de bits (dígito 1 o 0) y tienen un tipo asociado en la programación. Por ejemplo, un dato puede ser una simple letra (como "b") o un valor numérico (como 35).

DATOS

TIPO	DESCRIPCIÓN	EJEMPLO
Datos numéricos	Representan valores escalares de forma numérica y permiten realizar operaciones aritméticas comunes (+ - x /).	230 5000000
Datos alfanuméricos	Representan valores identificables de forma descriptiva. También pueden representar números, pero no es posible hacer operaciones matemáticas con ellos y van entre comillas.	Texto v@lor3s texto 12345
Datos lógicos	Solo pueden tener dos valores (verdadero o falso), ya que representan el resultado de la comparación entre otros datos (numéricos o alfanuméricos).	1+0=0 Falso 1+1=1 Verdadero 0+1=0 Falso 2>3= 0=Falso 3>2=1= Verdadero

Tabla 1. Lista comparativa de datos simples que podemos utilizar en el desarrollo de código de ejemplo.

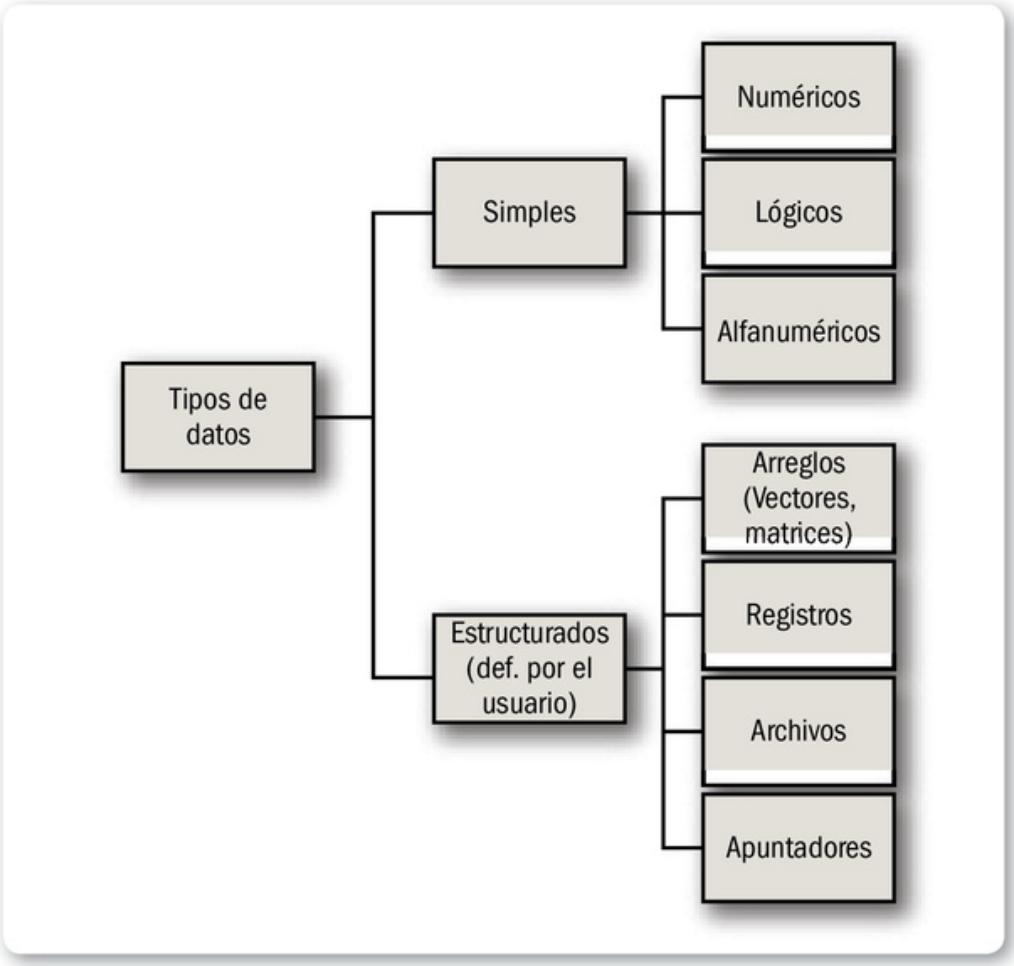


Figura 3. Tipos de datos. La categoría del dato determina la naturaleza del conjunto de valores que puede tomar una variable.

Qué son y cómo se usan las variables

Los nombres que representan el valor de un dato, ya sea numérico o alfanumérico, son variables. En esencia, una variable es un espacio en

NORMAS GENERALES PARA CREAR VARIABLES

Para dar nombres a las variables, debemos saber que: pueden tener hasta 40 caracteres, deben empezar obligatoriamente con una letra (a-z o A-Z), no pueden contener espacios en blanco, el resto de los dígitos pueden ser números, y es posible incluir caracteres especiales (como el guión o el punto).

la memoria de la computadora que permite almacenar temporalmente un dato durante la ejecución de un proceso, y cuyo contenido puede cambiar mientras corre un programa.

Para utilizar una variable, debemos darle un nombre con el cual identificarla dentro de un algoritmo. Si fuera un lenguaje de programación, este nombre apuntaría automáticamente a un espacio de memoria. Tanto en pseudocódigo como en un programa, es posible crear tantas variables como sean necesarias. Así, por ejemplo, podemos crear:

- **A = 100**: Variable tipo numérica A cuyo valor es 100.

Ciudad = "Córdoba": Variable alfanumérica o de tipo carácter Ciudad, cuyo valor es "Córdoba"

- **A = C + B**: Variable numérica A cuyo valor es la suma de los valores de las variables numéricas C y B. Es una variable calculada.

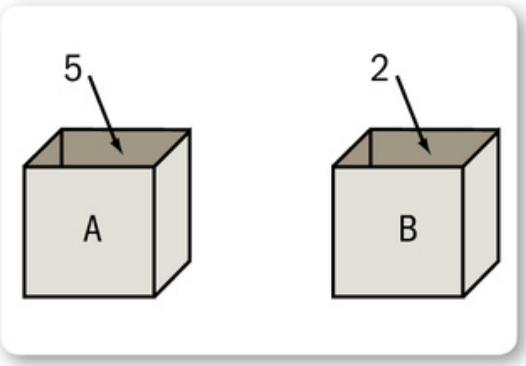


Figura 4. Aquí el espacio de memoria se representa por cajas, y se asignan valores a las variables (A=5 y B=2).

Como vemos en los ejemplos, el valor que les damos a las variables se llama **asignación**. Se trata del proceso que tendremos que efectuar cuando queramos grabar o hacer una operación aritmética. La asignación consiste en el paso de valores a una zona de la memoria, que pueden ser las variables. Dicha zona será reconocida con el nombre de la variable que recibe el valor, y se puede clasificar de la siguiente forma:

- **Simples**: consiste en pasar un valor constante a una variable. Dos ejemplos: a <-- 15 ; a = 15
- **Contador**: sirve para verificar el número de veces que se realiza un proceso. Dos ejemplos: a <-- a + 1 ; a = a + 1
- **Acumulador**: se utiliza como un sumador en un proceso. Dos ejemplos: a <-- a + b ; a = a + b
- **De trabajo**: recibe el resultado de una operación que involucre muchas variables. Dos ejemplos: a <-- c + b * 2 / 4 ; a = c + b * 2 / 4

Nota: Por lo general, en el pseudocódigo que escribimos en papel

se utiliza el símbolo de asignación <--. También podemos usar el igual (=) para representar esta acción.

Clasificación de las variables

Ahora que ya hemos visto lo que significa la asignación de valores en variables, pasemos a estudiar a fondo la clasificación y las características de estas.

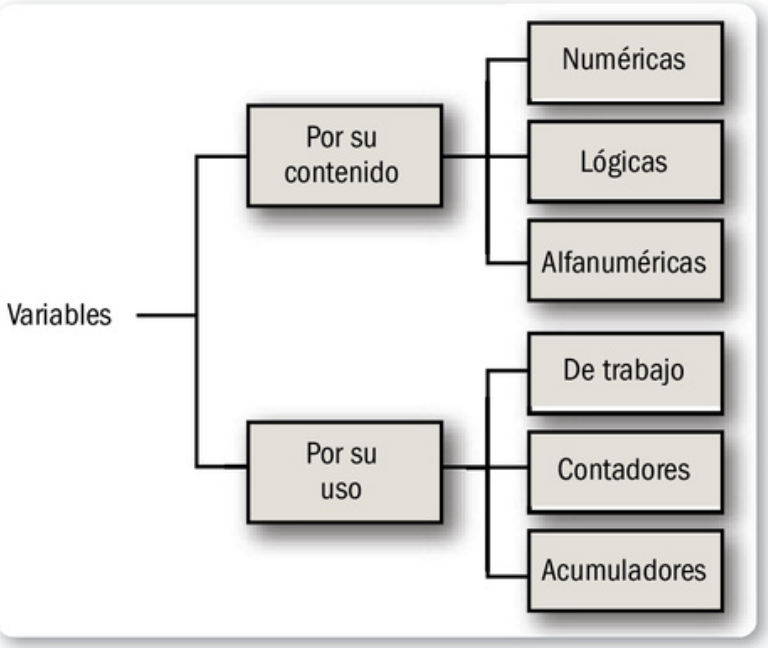


Figura 5. La clasificación de las variables se determina en función de su contenido y uso.

En la **Tabla 2** vemos las variables que podemos **crear por su contenido**, junto con su descripción y ejemplo.

CONSTANTE

Las constantes son declaraciones de datos a las que se les asigna un espacio en la memoria para su almacenamiento y no cambian durante la ejecución del programa. Estas se definen durante el tiempo de la complilación, y pueden ser tanto numéricas como alfanumércas.

CONTENIDO

VARIABLES Y SU CONTENIDO	DESCRIPCIÓN	EJEMPLO
Variables numéricas	Almacenan valores numéricos (positivos o negativos), es decir: números del 0 al 9, signos (+ y -) y el punto decimal.	Costo <- 2500 IVA <- 0.15 PI <- 3.1416
Variables lógicas	Solo pueden tener dos valores (cierto o falso), que representan el resultado de una comparación entre otros datos.	Habilitado <- 0 Habilitado <- 1
Variables alfanuméricas	Formadas por caracteres alfanuméricos (letras, números y caracteres especiales).	Letra <- 'a' Apellido <- 'ramos' Dirección <- 'Rondeau 165'

Tabla 2. Lista comparativa de los tipos de variables que existen en función de su contenido.

En la **Tabla 3** se muestran las variables que podemos crear **por su uso**, junto con su descripción y ejemplo.

USO

VARIABLES Y SU USO	DESCRIPCIÓN	EJEMPLO
Variables de trabajo	Reciben el resultado de una operación matemática completa y se usan normalmente dentro de un programa.	Ejemplo Suma = a + b /c
Contadores	Llevan el control del número cuando se realiza una operación o se cumple una condición, con los incrementos generalmente de uno en uno.	Contador = Contador +1
Acumuladores	Llevan la suma acumulativa de una serie de valores que se van leyendo o calculando progresivamente.	Acu = Acu + Calculo

Tabla 3. Lista comparativa de los tipos de variables en función de su uso.

Normas de escritura

Retomando las normas generales para escribir en pseudocódigo, debemos tener en cuenta cómo vamos a crear o declarar estas variables en nuestro programa. Como dijimos que las expresiones se asemejan al lenguaje humano, podemos utilizar la palabra "variable" para declarar una de ellas. Por ejemplo, es posible usar las siguientes formas:

Ejemplo 1:

```
INICIO
   Variable Nombre
   Variable Edad
   Nombre <-- "Juan"
   Edad <-- 20
   Mostrar Nombre Y Edad
FIN
```

Como podemos observar en el ejemplo 1, no solo creamos o declaramos la variable y le dimos un nombre, sino que también especificamos qué tipo de dato se puede almacenar en ella y, luego, le asignamos el valor. El modelo que utilicemos dependerá del nivel de trabajo que queramos realizar. Lo mejor sería dejar todo detallado en el pseudocódigo, para así, después, pasarlo al lenguaje de programación sin mayores inconvenientes. Por ejemplo, la sintaxis sugerida sería:

Variable Nombre_de_la_variable tipo_de_dato

Ejemplos válidos de nombres de variables:

Variable FechaNueva

Variable H123

INDENTACIÓN

Dentro de los lenguajes de programación que se aplican a las computadoras, la indentación representa un tipo de notación secundaria que se utiliza para mejorar la legibilidad del código fuente por parte de los programadores, teniendo en cuenta que los compiladores o intérpretes raramente consideran los espacios en blanco entre las sentencias de un programa.

Variable cantidad_de_Alumnos
Variable Pedido.Almacen

Ejemplos NO válidos de nombres de variables:
Variable 1contador
Variable primer-valor N

Algunos lenguajes de programación deben tener declaradas las variables que se van a utilizar en todo el programa. De esta forma, al comenzar el programa, estarán declarados: nombre, tipo (numérica o alfanumérica) y valor inicial. Las variables también pueden **inicializarse** dándoles un valor inicial. Por defecto, todas las variables para las que no especifiquemos un valor inicial valen **cero** si son de tipo numérico y **nulo** si son de tipo carácter/texto. Cabe destacar que el tipo de dato nulo no es cero ni espacio en blanco, es nulo (una traducción podría ser vacío).

Es importante conocer cómo se utilizan las variables y qué combinaciones de operaciones podemos realizar con ellas. El próximo tema nos dará una amplia visión sobre este aspecto.

Cómo se utilizan los operadores

En todos los casos en que precisemos realizar desarrollos para solucionar algún inconveniente, nos veremos involucrados en la necesidad de efectuar operaciones de distintos tipos: suma, resta, concatenación, procesos lógicos, etc. Estos elementos se relacionan de modo diferente, con valores de una o más variables y/o constantes.

A continuación, veremos los operadores que podemos utilizar en el pseudocódigo para manipular valores.

IDENTIFICADORES

Los identificadores son aquellos que representan los datos de un programa, las constantes, las variables y los tipos de datos. Se trata de una secuencia de caracteres que se utilizan para identificar una posición en la memoria de la computadora y obtener así el acceso a su contenido. A modo de ejemplo, podemos mencionar: Nombre; Numero_horas; Calificacion.

Aritméticos

Los operadores aritméticos permiten realizar operaciones matemáticas con los valores de variables (suma, resta, multiplicación, etcétera), y pueden usarse con datos enteros o reales.

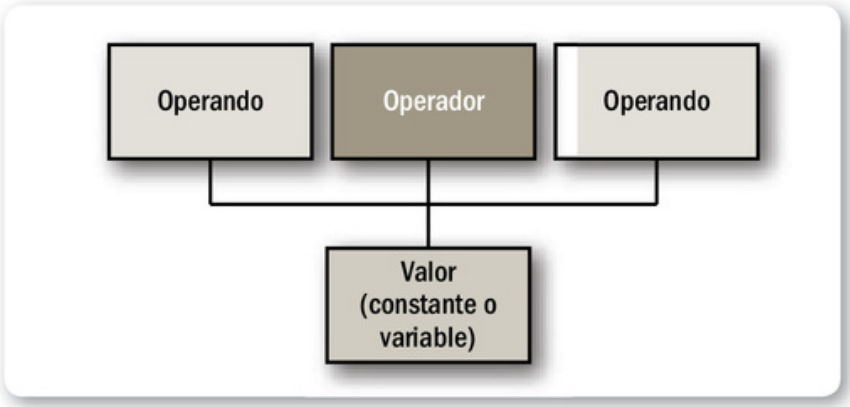


Figura 6. En este esquema podemos ver la representación de una fórmula sobre operadores aritméticos.

ARITMÉTICOS

SIGNO	SIGNIFICADO
+	Suma
-	Resta
*	Multiplicación
/	División
^	Potenciación
MOD	Resto de la división entera

Tabla 4. En este listado podemos ver los signos aritméticos que podemos utilizar en programación, junto a sus respectivos significados.

Ejemplos:	Expresión	Resultado
	7 / 2	3.5
	4 + 2 * 5	14

Es importante tener en cuenta la **prioridad de los operadores aritméticos**. Todas las expresiones entre paréntesis siempre se evalúan primero. Aquellas con paréntesis anidados se evalúan desde adentro hacia afuera (el paréntesis más interno se evalúa primero).

Dentro de una misma expresión, los operadores se evalúan en el siguiente orden:

1. ^ Potenciación
2. *, /, mod Multiplicación, división, módulo
3. +, - Suma y resta

Los operadores en una misma expresión y con igual nivel de prioridad se evalúan de izquierda a derecha.

Ejemplos:

4 + 2 * 4 = 12

23 * 2 / 5 = 9.2

3 + 5 * (10 - (2 + 4)) = 23

2.1 * (1.5 + 12.3) = 2.1 * 13.8 = 28.98

Lógicos

Los operadores lógicos se utilizan para establecer relaciones entre valores lógicos, que pueden ser el resultado de una expresión relacional. Dentro del pseudocódigo, por lo general pueden tomar dos valores para indicar su estado:

1 - Verdadero – True

0 - Falso - False

NULO/NULA

En el camino del desarrollo, nos encontraremos con diferentes tipos de datos que aceptan tener el valor NULO. Pueden tener referencias como NULL o null, dependiendo del lenguaje. Esto se utiliza para indicar que el tipo de dato no tiene ningún valor asignado y, frecuentemente, se aplica en bases de datos.

Los tipos de operadores lógicos que podemos aplicar a la programación son los siguientes:

And - Y

Or - O

Not - Negación - No

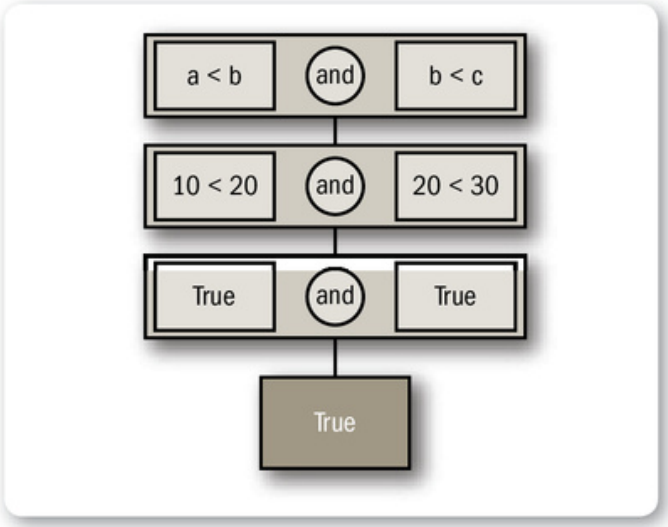


Figura 7. Aquí T significa verdadero, y F, falso. Las variables son a=10, b=20 y c=30.

Podemos ver que tanto la primera expresión como la segunda son verdaderas y, por eso, el resultado final también lo es. Las posibles combinaciones lógicas que encontraremos con los diferentes operadores se muestran en las siguientes tablas.

AND

CONDICIÓN1	OPERADOR	CONDICIÓN2	RESULTADO
Verdadero	AND	Verdadero	Verdadero
Verdadero		Falso	Falso
Falso		Verdadero	Falso
Falso		Falso	Falso

Tabla 4. En la aplicación del operador **AND**, el hecho de que alguna de las condiciones sea falsa hará que el resultado también lo sea.

Supongamos que creamos las variables **EDAD** y **ALTURA** y, en la primera parte, preguntamos en pseudocódigo si EDAD es mayor que 18 Y su ALTURA es menor que 1.70. Esta expresión devolverá verdadero solo si ambas son verdaderas. (Edad > 18) Y (Altura < 1.70)

Por ejemplo, veamos qué sucede si las variables toman los siguientes valores:

Edad <- 21	Edad <- 12	Edad <- 21
Altura <- 1.90	Altura <- 1.90	Altura <- 1.50
El resultado sería verdadero	**El resultado sería falso**	**El resultado sería falso**

El operador **OR** u **O** se utiliza para preguntar sobre el cumplimiento de una condición u otra; el resultado será verdadero siempre y cuando alguna expresión también lo sea.

OR

CONDICIÓN1	OPERADOR	CONDICIÓN2	RESULTADO
Verdadero	OR	Verdadero	Verdadero
Verdadero		Falso	Verdadero
Falso		Verdadero	Verdadero
Falso		Falso	Falso

Tabla 5. En la aplicación del operador **OR**, el resultado solo será falso si ambas condiciones son falsas.

ÁLGEBRA DE BOOLE

Hace referencia a una estructura algebraica que esquematiza las operaciones lógicas Y, O, NO y SI (AND, OR, NOT, IF), así como el conjunto de operaciones unión, intersección y complemento. Encontraremos que los datos booleanos o lógicos influyen de gran manera en la informática y la electrónica.

Supongamos que creamos una variable **EstadoCivil** y **Sueldo** y, en la primera parte, preguntamos en pseudocódigo si el estado civil es igual a C (entendiéndose que C es casado) o su Sueldo es mayor que 2000. Esta expresión devolverá verdadero si alguna de las dos es verdadera.

(EstadoCivil = 'C') o (Sueldo > 2000)

EstadoCivil <- 'C'	EstadoCivil <- 'S'	EstadoCivil <- 'S'
Sueldo <- 1000	Sueldo <- 2100	Sueldo <- 1500
El resultado sería verdadero	**El resultado sería verdadero**	**El resultado sería falso**

Como podemos ver en la **Tabla 6**, el operador **Not** o **NO** se utiliza para preguntas de negación en las condiciones deseadas.

NO

CONDICIÓN1	OPERADOR	RESULTADO
Verdadero	**NO**	**Falso**
Falso		**Verdadero**

Tabla 6. Con el operador **NO** el resultado invierte la condición de la expresión.

Supongamos que creamos una variable **sexo** y queremos preguntar por aquellas variables que NO son femenino.

NO (sexo = "Femenino")

sexo <- "Masculino"	sexo <- "Femenino"
El resultado sería verdadero	**El resultado sería falso**

NORMA

Es un término que proviene del latín y significa "escuadra". Una norma es una regla que debe ser respetada y que permite ajustar ciertas conductas o actividades. En el ámbito informático, es lo que aplicamos en el orden de la programación, estructuras, declaraciones, funciones, etc.

PRIORIDADES

OPERADORES LÓGICOS	SIGNIFICADO	OPERADORES EN GENERAL
1. Not	Negación (NO)	1. ( )
2. And	Producto lógico (Y)	2. ^
3. Or	Suma lógica (O)	3. *, /, Mod, Not
		4. +, -, And
		5. >, <, > =, < =, < >, =, Or

Tabla 7. Prioridades en la resolución que debemos tener en cuenta para las operaciones aritméticas

Relacionales

Se utilizan para establecer una relación entre dos valores. Al comparar estos valores entre sí, se produce un resultado verdadero o falso. Los operadores relacionales comparan valores del mismo tipo, numéricos o cadenas. Estos tienen igual nivel de prioridad en su evaluación.

RELACIONALES

OPERADOR	COMPLEMENTO	OPERADOR	COMPLEMENTO
<	> =	Menor que	Mayor o igual que
<=	>	Menor o igual que	Mayor que
>	<=	Mayor que	Menor o igual que
>=	<	Mayor o igual que	Menor que
=	¹	Igual que	Distinto de (diferente)
¹	=	Distinto de (diferente)	Igual que

Tabla 8. Los operadores relacionales tienen menor prioridad que los aritméticos.

Ejemplo	Resultado
25 <= 25	Verdadero
25 <> 25	Falso
25 <> 4	Verdadero
50 <= 100	Verdadero
500 >= 1	Verdadero
1 = 6	Falso

Ejemplos no lógicos:

- a < b < c
- 10 < 20 < 30
- T > 5 < 30
- (no es lógico porque tienen diferentes operandos)

Cuando se comparan caracteres alfanuméricos, se lo hace de uno en uno, de izquierda a derecha. Si las variables son de distinta longitud, pero exactamente iguales, se considera que la de menor longitud es menor. Los datos alfanuméricos son iguales si y solo si tienen la misma longitud y los mismos componentes. Las letras minúsculas son mayores que las mayúsculas, y cualquier carácter numérico es menor que cualquier letra mayúscula o minúscula.

Teniendo en cuenta lo explicado anteriormente, a continuación trabajaremos con algunos ejemplos que nos ayuden a comprenderlo mejor. Para eso, es importante tener en cuenta la siguiente sintaxis de prioridad, que es con la que nosotros trabajaremos:

carácter numérico < mayúsculas < minúsculas

CADENA

Una cadena o string es una sucesión de caracteres que se encuentran delimitados por comillas (“”). La longitud de la cadena es la cantidad de caracteres que la forma, incluyendo los espacios, que son un carácter más. Por ejemplo: **“Sudamérica, Argentina”** es una cadena de longitud 21.

Comparación	Resultado
"A" < "B"	Verdadero
"AAAA" > "AAA"	Verdadero
"B" > "AAAA"	Verdadero
"C" < "c"	Verdadero
"2" < "12"	Falso

Asignación de valores

Como vimos anteriormente, para que las variables tomen un valor, debemos asignárselo en pseudocódigo por medio de = o <--. La cooperación de asignación le permite a la computadora evaluar una expresión matemática y almacenar el resultado final en una determinada variable. La sintaxis que podemos utilizar para la asignación es la siguiente:

Nombre_de_la_variable <-- expresión o variable

ASIGNACIÓN		
▼ ASIGNACIÓN ARITMÉTICA	▼ ASIGNACIÓN LÓGICA	▼ CARACTERES O CADENA DE CARACTERES
Variable var1, var2, var3 tipo numérico	Variable var1, var2, var3 tipo lógico	Variable vCad, car tipo texto
var1 <-- 3+4*2	var1 <-- 5 < 2	car <-- 'S'
var2 <-- 0.65 / 0.2	var2 <-- 7 >= 3	vCad <-- "25 de diciembre de 1998"
var3 <-- var1 / var2	var3 <-- var1 o var2	

Tabla 9. En la asignación a variables, el símbolo <-- indica que el valor de la parte derecha del enunciado se le asigna a la variable de la izquierda.

Para la construcción de un enunciado de asignación, es preciso considerar ciertas **reglas**. En primer lugar, toda variable del lado derecho debe estar definida y solo la de la izquierda puede cambiar de valor cuando antes tenía un valor asignado. Las variables del lado derecho siempre conservan su valor, aunque es importante tener en cuenta que, si la variable de la parte izquierda está también en la derecha, esta cambia de valor por aparecer en la izquierda.

Hasta aquí, hemos visto: el inicio de la codificación en pseudocódigo, el uso de variables, sus diferentes utilidades y la asignación de valores. A continuación, en la **Figura 8**, veremos un esquema que nos ilustrará, mediante un ejemplo sencillo, cómo se confeccionaría un pseudocódigo para presentar en un caso concreto.

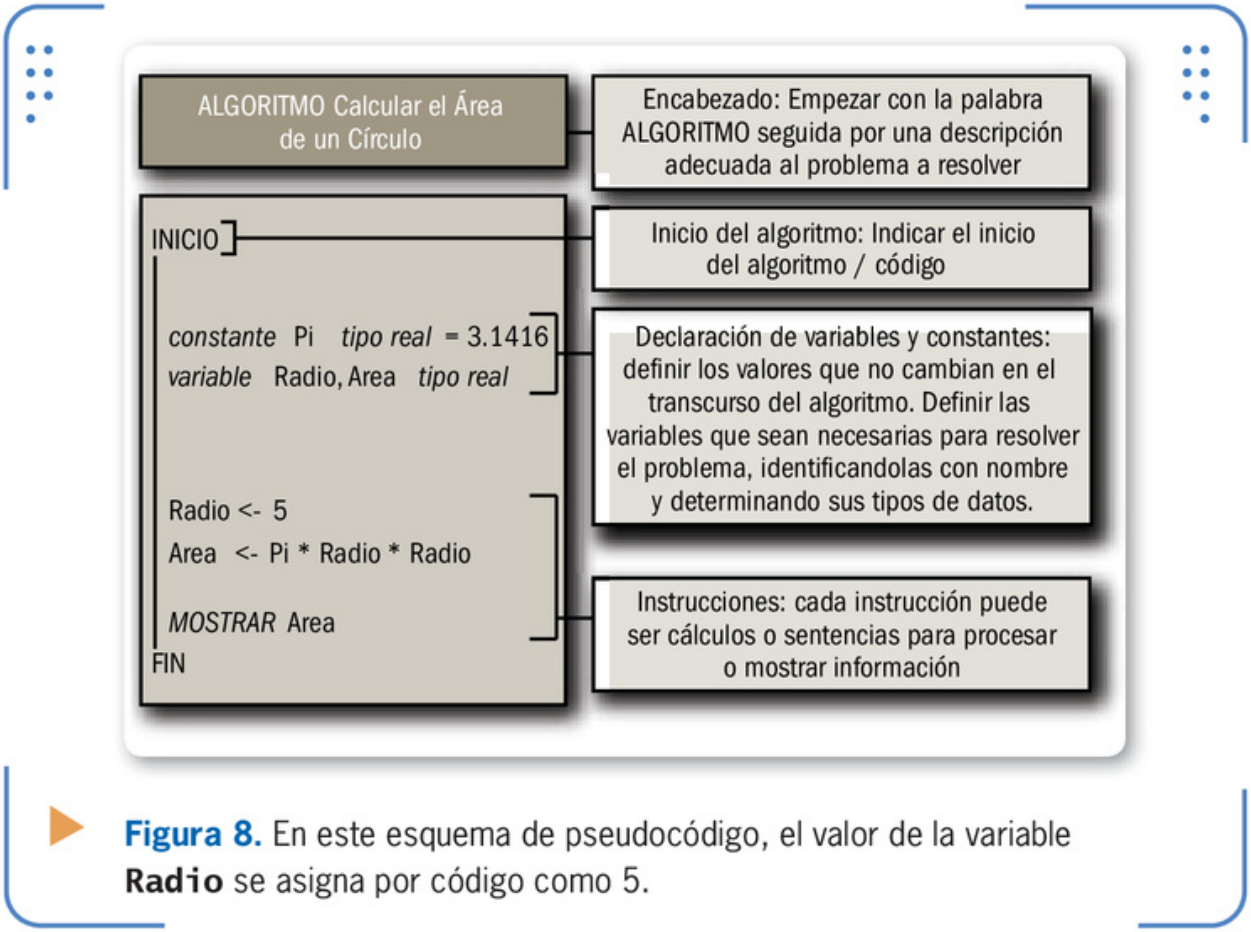


Figura 8. En este esquema de pseudocódigo, el valor de la variable **Radio** se asigna por código como 5.

Para capturar un valor que el usuario pueda ingresar, podemos utilizar las palabras **LEER** o **MOSTRAR**. En el caso de **LEER**, capturaremos un valor para la aplicación; mientras que **MOSTRAR** dará un resultado. Veámoslo en el ejemplo de la **Figura 9**.

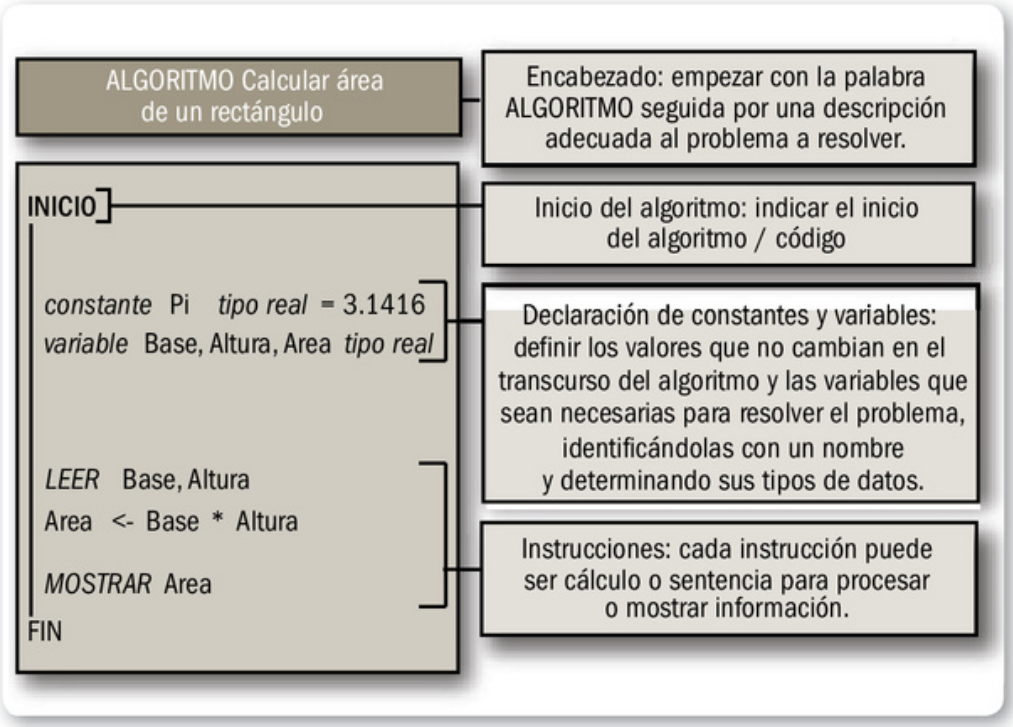


Figura 9. Esquema de pseudocódigo. Aquí vemos el detalle que podemos estructurar para nuestros ejemplos de algoritmo o código por desarrollar.

Ahora pasemos a un ejemplo más específico, donde debamos hacer un algoritmo que nos permita calcular el sueldo básico de una persona.

Signo	Significado
+	Concatenación
&	Concatenación

Para esto, debemos ingresar la tarifa horaria y las horas trabajadas.

Expresión	Resultado
"Pseudo" + "código"	"Pseudocódigo"
"3" + "4567"	"34567"
"Hola " + "que tal ?"	"Hola que tal ?"

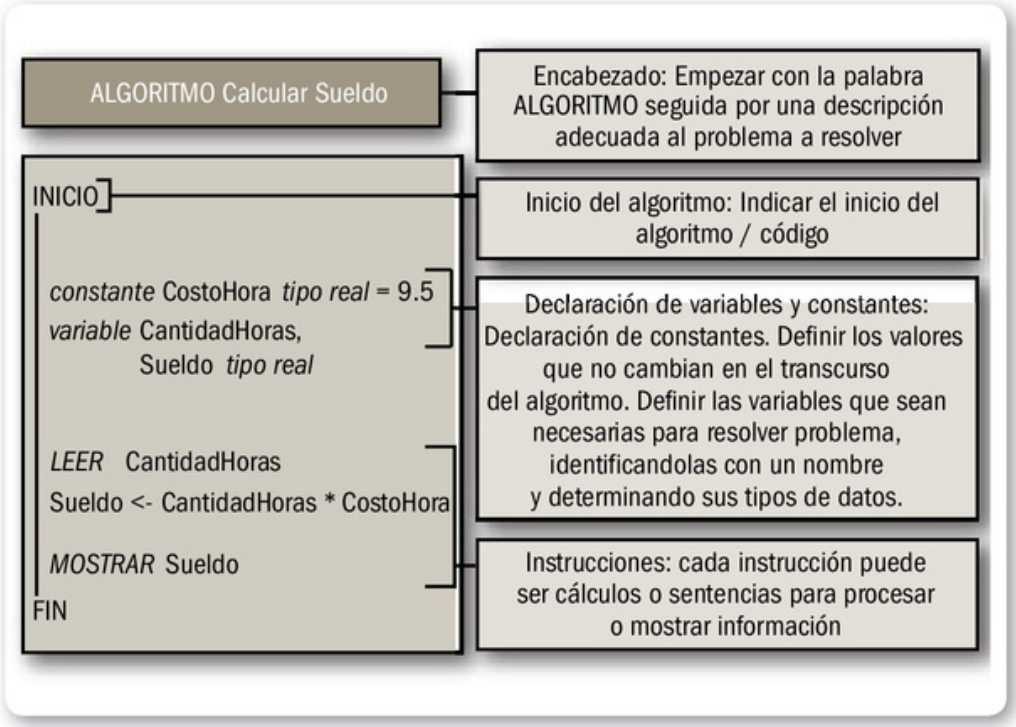


Figura 10. En este esquema de pseudocódigo, la fórmula del sueldo es: Sueldo= Costo hora x Cantidad de horas.

Cuando tratamos el tema de las asignaciones, debemos tener en cuenta los símbolos + e & que utilizaremos para unir o concatenar datos. Dependiendo del tipo de lenguaje que vayamos a utilizar, usaremos el símbolo indicado para concatenar. En este ejemplo que estamos trabajando, usaremos el +.

En estos ejemplos, vemos operaciones simples que podemos confeccionar en pseudocódigo, y otros casos más complejos, en los cuales debemos tener en cuenta las estructuras que podemos utilizar. A continuación, vamos a encontrar el desarrollo completo sobre ellas.

ASIGNACIÓN DESTRUCTIVA

Cuando decimos que toda asignación es destructiva, significa que el valor previo que tiene la variable se pierde, y se reemplaza por el nuevo valor que asignamos. Así, cuando se ejecuta esta secuencia:
B <- 25; B <- 100; B <- 77, el valor final que toma B será 77, ya que los valores 25 y 100 son destruidos.

Entrada y salida de información

Para procesar los datos que vamos a obtener del usuario, debemos asignarlos a variables. Para esto, utilizamos la instrucción **LEER** o, también, **INGRESAR**. Por ejemplo:

```
variable varNumero tipo numero
LEER varNumero
```

Dicha instrucción le pide al usuario que ingrese un valor que luego será asignado a la variable **varNumero**.

```
variable Edad, Peso tipo numero
variable Sexo tipo texto
LEER Edad, Peso, Sexo
```

Esto representa la lectura de tres valores que se van a almacenar en las variables Edad, Peso y Sexo.

Con el código anterior, hemos capturado información para nuestro programa utilizando pseudocódigo. Cuando deseamos mostrar un resultado en un mensaje, debemos aplicar la instrucción **IMPRIMIR** O **MOSTRAR**, como vemos en el siguiente ejemplo:

```
IMPRIMIR "Hola" // MOSTRAR "Hola"
```

Cuando en pseudocódigo queremos mostrar en pantalla el mensaje "Hola", debemos recordar que la palabra tiene que ir entre comillas, porque pertenece a una cadena de texto.

```
variable A tipo numero <-- 520
IMPRIMIR A // MOSTRAR A
```

De esta forma, podemos mostrar en la pantalla el valor que está almacenado en la variable A; en este caso: el número 520.

```
variable A, B, Promedio tipo numero
A <-- 15
B <-- 7
Promedio <-- (A + B) / 2
```

```
IMPRIMIR "El valor del promedio es:", Promedio
//MOSTRAR "El valor del promedio es:", Promedio
```

Esta instrucción muestra el mensaje que está entre comillas y, luego, el valor de la variable promedio. La coma separa el mensaje de la variable, y el resultado de promedio es 11. De este modo, lo que se verá en pantalla será: El valor del promedio es: 11

Hemos visto dos comandos que vamos a utilizar en nuestros pseudocódigos: **LEER** y **MOSTRAR** / **IMPRIMIR**. También tenemos la posibilidad de mostrar un mensaje cuando le solicitamos algún dato al usuario, por medio del comando **LEER**:

```
variable edad tipo numero <-- 0
LEER "Ingrese su edad", edad
```

El valor de la variable que le pedimos al usuario se asigna a edad. Esta instrucción se verá de la siguiente forma en la pantalla:

```
Ingrese su edad ?
```

> ES IMPORTANTE DECLARAR CORRECTAMENTE LAS VARIABLES Y TIPOS DE DATOS

Hasta aquí hemos visto de qué modo debemos actuar para declarar y utilizar las variables con sus tipos de datos, y cómo podemos aplicar los diferentes tipos de operadores. También vimos cómo deben tomarse los valores ingresados por los usuarios y qué opciones podemos elegir para mostrar la información. A continuación, aprenderemos a realizar la parte "inteligente", que nos permitirá resolver diferentes situaciones que puedan presentarse en la programación.

BENEFICIOS DEL PSEUDOCÓDIGO

En comparación con los diagramas de flujo, el pseudocódigo permite representar fácilmente las operaciones repetitivas complejas, agilizar el pasaje de pseudocódigo a lenguaje de programación formal, y mostrar los niveles y estructuras gracias a la indentación. También mejora la claridad de la solución de un problema, ya que da como correcta la opción más conveniente.

Todo tiene un orden en la programación

El funcionamiento del equipo se basa en la ejecución de los comandos a medida que va leyendo el archivo (de arriba hacia abajo), hasta alcanzar un comando que lo dirija hacia una ubicación específica del programa. Para que este trabajo se realice correctamente, es importante que la información esté organizada y estructurada de forma adecuada. De esta forma, podrá obtenerse un rendimiento razonable en la memorización, tratamiento y recuperación de esa información.

Estructuras de control

Las estructuras de operación de programas constituyen un grupo de formas de trabajo que, mediante el manejo de variables, nos permiten realizar ciertos procesos específicos para solucionar los problemas.

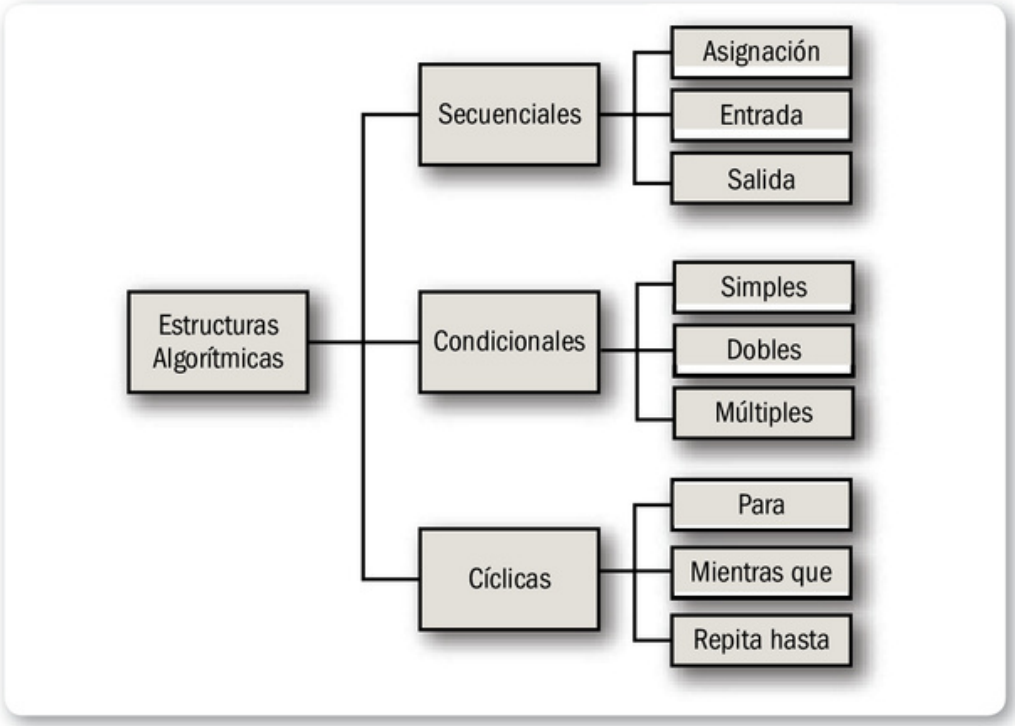


Figura 11. En esta imagen podemos ver que las estructuras de control se clasifican de acuerdo con su complejidad.

Secuencial

Las estructuras secuenciales son todas aquellas estructuras que estuvimos utilizando en los casos anteriores. Se trata de un número definido de instrucciones que se ubican en un orden específico y se suceden una tras otra.

Condicional

En este caso, se compara una variable con otros valores, para que, sobre la base del resultado, se siga un curso de acción dentro del programa. Cabe mencionar que la comparación puede hacerse contra otra variable o contra una constante, según sea necesario. Existen tres tipos básicos: simples, dobles y múltiples.

Figura 12. En la decisión simple de comer una manzana o una naranja, la elección da como resultado la acción de "comer".

Simples

Las estructuras condicionales simples se conocen como **toma de decisión** y tienen la siguiente sintaxis:

```
Si <condición> entonces
    Instrucción/es
Fin Si
```

Si: indica el comando de comparación.
Condición: indica la condición por evaluar.
Instrucción: son las acciones que van a realizarse cuando se cumple o no la condición.

Dentro del pseudocódigo, podemos encontrar el siguiente ejemplo: debemos preguntar si la edad de una persona es mayor o igual que 18 años; si esto se cumple, mostramos un mensaje que diga "ES MAYOR". Veamos el código:

```
INICIO
   Variable edad tipo numero
   edad <-- 15

   //podemos utilizar LEER edad, si deseamos que un usuario ingrese por teclado
      el valor
   Si edad>=18 entonces
   MOSTRAR "Es mayor"
   Fin Si
FIN
```

Nota: por lo general, en las anotaciones en forma de comentario, tanto en pseudocódigo como en algunos lenguajes de programación, podemos encontrar los signos de barras, " // ".

Si deseamos comparar valores de variables, podemos considerar el siguiente ejemplo. Contamos con la altura de dos personas: A y B. Si el más alto es A, mostramos un mensaje. Veamos el código:

```
INICIO
   Variable alturaA, alturaB tipo real
   alturaA<-- 1.5
   alturaB<-- 1.9
```

```
    //podemos utilizar LEER edad, si deseamos que un usuario ingrese por teclado
        los valores

    Si alturaA>= alturaB entonces
        MOSTRAR "La persona más alta es A"
    Fin Si
FIN
```

Otro ejemplo que podemos tomar para un cálculo con variables es: sumamos dos valores y, si el resultado de la operación es mayor que 50, informamos que "El valor es ALTO". Veamos el código:

```
INICIO
    Variable numeroA, numeroB tipo numero
    numeroA <-- 15
    numeroB <-- 20

    //podemos utilizar LEER edad, si deseamos que un usuario ingrese por teclado
        los valores

    Si (numeroA + numeroB) > 50 entonces
        MOSTRAR "El valor es ALTO"
    Fin Si
FIN
```

Doble

Las estructuras condicionales dobles permiten elegir entre dos opciones, en función del cumplimiento o no de una determinada condición. Tienen la siguiente sintaxis:

Si <condición> **entonces**
Instrucción/es
Sino
Instrucción/es
Fin Si

Si: indica el comando de comparación.

Condición: indica la condición que se va a evaluar.

Entonces: precede a las acciones por realizar cuando se cumple la condición.

Instrucción: son las acciones que se realizarán cuando se cumple o no la condición.

Sino: precede a las acciones por realizar cuando no se cumple la condición. Dependiendo de si la comparación es cierta o falsa, es posible realizar una o más acciones.

A continuación, utilizaremos los ejemplos anteriores, pero aplicados a esta estructura. Por ejemplo: si la edad es mayor o igual que 18, mostraremos: "es mayor de edad"; en caso contrario: "es menor de edad":

```
INICIO
    Variable edad tipo numero
    edad <-- 15

    Si edad>=18 entonces
        MOSTRAR "Es mayor de edad"
    Sino
        MOSTRAR "Es menor de edad"
    Fin Si
FIN
```

Comparando dos valores de variables (por ejemplo: las alturas de los dos sujetos A y B), si el más alto es A debemos mostrar: "La persona más alta es A"; en caso contrario: "La persona más alta es B". Veamos el código:

MOSTRAR Y OBTENER

En pseudocódigo, encontramos diferentes palabras que expresan la misma acción. Por ejemplo, para mostrar datos podemos utilizar: **MOSTRAR**, **ESCRIBIR**, **IMPRIMIR** o **PRINT**. En el caso de querer obtener datos, usamos **PEDIR** o **LEER**. Es recomendable emplear una sintaxis homogénea y unificar las palabras.

```
INICIO
    Variable alturaA, alturaB tipo real
    alturaA<-- 1.5
    alturaB<-- 1.9

    Si alturaA>= alturaB entonces
        MOSTRAR "La persona más alta es A"
    Sino
        MOSTRAR "La persona más alta es B"
    Fin Si
FIN
```

A veces, cuando realizamos un cálculo con variables, debemos alterar nuestro mensaje en función del resultado. Por ejemplo: sumamos dos valores y, si es mayor que 50, informamos: "El valor es ALTO"; en caso contrario: "El valor es BAJO". Veamos el código:

```
INICIO
    Variable numeroA, numeroB tipo numero
    numeroA <-- 15
    numeroB <-- 20

    //podemos utilizar LEER edad, si deseamos que un usuario ingrese por teclado
        los  valores

    Si (numeroA + numeroB) > 50 entonces
        MOSTRAR "El valor es ALTO"
    Sino
        MOSTRAR "El valor es BAJO"
    Fin Si
FIN
```

El siguiente código comprobará que, al introducir un número por teclado, nos diga si es positivo o negativo:

```
INICIO
   Variable Num tipo numero<-- 0
   ESCRIBIR "Escriba un número: "
   LEER Num
   SI Num >= 0 ENTONCES
      MOSTRAR "Es positivo"
   SINO
      MOSTRAR "Es negativo"
   FINSI
FIN
```

El siguiente código comprobará que, al introducir un número por teclado, nos diga si es par o impar:

```
INICIO
   Variable Num tipo numero<-- 0
   ESCRIBIR "Escriba un número: "
   LEER Num
   SI num = int( num / 2 ) * 2  ENTONCES
      MOSTRAR "Es par"
   SINO
      MOSTRAR "Es impar"
   FINSI
FIN
```

Múltiples o anidadas

Estas estructuras de comparación son decisiones especializadas que nos permiten comparar una variable con distintos resultados posibles, ejecutando una serie de instrucciones específicas para cada caso. Estas tienen la siguiente sintaxis:

```
   Si <condición> entonces
      Instrucción/es
   Sino
         Si <condición> entonces
            Instrucción/es
```

```
    Sino
        Si <condición> entonces
            Instrucción/es
        Sino
            ... y así sucesivamente...
        Fin Si
    Fin Si
Fin Si
```

Necesitamos realizar un algoritmo que pida la altura de una persona. Para eso, vamos a establecer que: si la altura es menor o igual que 150 cm, mostrará el mensaje: "Persona de altura baja"; si la altura está entre 151 y 170, mostrará: "Persona de altura media"; y si la altura es mayor que 171, mostrar el mensaje: "Persona alta".

```
INICIO
    Variable Altura tipo numero
    ESCRIBIR "¿Cuál es tu altura?: "
    LEER Altura
    Si Altura <=150 entonces
        ESCRIBA "Persona de altura baja"
    Sino
        Si Altura <=170 entonces
            ESCRIBA "Persona de altura media"
        Sino
            Si Altura>170 entonces
                ESCRIBA "Persona alta"
            Fin Si
        Fin Si
    Fin Si
FIN
```

Otra de las estructuras de comparación múltiple es una decisión especializada que nos permita evaluar una variable con distintos resultados posibles, ejecutando para cada caso una serie de instrucciones específicas. La sintaxis es la siguiente:

```
En_Caso_De <condición> haga

    Caso 1: Instrucción/es

    Caso 2: Instrucción/es

    Caso 3: Instrucción/es

        Sino
            Instrucción/es
Fin_Caso
```

También puede suceder que encontremos otra sintaxis para representar una misma estructura, Por ejemplo, en este caso, la sintaxis para el pseudocódigo sería la siguiente:

```
Segun_Sea <condición> hacer

   Caso 1:  Instrucción/es

    Caso 2:  Instrucción/es

    Caso 3:  Instrucción/es

        Sino
            Instrucción/es
Fin_Segun
```

Para ir cerrando con esta estructura, pademos a realizar un ejemplo sencillo que nos permita visualizar el concepto. En este caso, vamos a tener que diseñar la estructura necesaria para que el usuario pueda ingresar un valor numérico, establecido entre 1 y 5. Como respuesta, debemos asegurarnos de que el algoritmo muestre el mismo número, pero en formato de texto:

```
INICIO
   Variable num tipo numero

   ESCRIBIR "Ingrese un valor entre 1 y 5: "
   LEER num
      En_Caso_De num haga

         Caso 1:  MOSTRAR "Uno"

         Caso 2:  MOSTRAR "Dos"

         Caso 3:  MOSTRAR "Tres"

         Caso 4:  MOSTRAR "Cuatro"

         Caso 5:  MOSTRAR  "Cinco"

      Sino
         MOSTRAR "No ingreso un valor entre 1 y 5"
   Fin_Caso

FIN
```

Otro ejemplo puede ser que el usuario ingrese un número entre 1 y 7, y el algoritmo deba dar como resultado su correspondiente día de la semana. Por ejemplo: 1- Lunes; 2- Martes; 3- Miércoles; 4- Jueves; 5- Viernes; 6- Sábado; 7- Domingo.

```
INICIO
   Variable num tipo numero

   ESCRIBIR "Ingrese un valor entre 1 y 7: "
   LEER num
      En_Caso_De num haga

         Caso 1:  MOSTRAR "Lunes"
```

```
            Caso 2:  MOSTRAR "Martes"

            Caso 3:  MOSTRAR "Miércoles"

            Caso 4:  MOSTRAR "Jueves"

            Caso 5:  MOSTRAR  "Viernes"

            Caso 6:  MOSTRAR  "Sábado"

            Caso 7:  MOSTRAR  "Domingo"

        Sino
            MOSTRAR "No ingreso un valor entre 1 y 7"
    Fin_Caso

FIN
```

Hasta aquí, hemos visto las estructuras de controles utilizadas para preguntar secuencialmente sobre alguna condición o caso. Es importante tener en cuenta que podemos utilizar las sentencias **SI – FinSi** para condiciones simples de una sola respuesta, y **SI-Sino-FinSi** para condiciones dobles con dos posibles respuestas. También podemos anidar las expresiones **SI**, una dentro de otra, con el fin de resolver condiciones complejas o que deben ir cumpliéndose una dentro de otra. Por último, es posible usar **Segun_Sea**, siempre que queramos preguntar por una determinada condición y, dependiendo de su valor o estado, realizar cierta acción.

Repetitivas o estructuras cíclicas

Se utilizan estructuras repetitivas o cíclicas en aquellas situaciones cuya solución necesita un mismo conjunto de acciones, que se puedan ejecutar una cantidad específica de veces. Esta cantidad puede ser fija, si fuese previamente determinada por el desarrollador; o variable, si actuara en función de algún dato dentro del programa.

Figura 13. Para pintar una pared, debemos repetir una acción hasta cumplir la condición: pasar la brocha hasta cubrir la superficie total.

Las estructuras repetitivas o cíclicas se clasifican en:

- **Ciclos con un número determinado de iteraciones**

Para – hasta – paso - hacer: son aquellos en que el número de iteraciones se conoce antes de ejecutarse el ciclo. La forma de esta estructura es la siguiente:

```
Para <variable> <expresión1> hasta <expresión2> paso <expresión3> hacer
    Instrucción/es
Fin_Para
```

Dado un valor inicial **expresión1** asignado a la **variable**, esta se irá aumentando o disminuyendo de acuerdo con la **expresión3** hasta llegar a la **expresión2**. En el caso de omitir el paso, eso va a significar que la variable aumentará de uno en uno.

Veamos un ejemplo aplicando esta estructura: realizar un algoritmo que muestre los números de uno en uno hasta 10:

```
INICIO

   Variable contador tipo numero <-- 0
      Para contador <-- 1 hasta 10 paso 1 hacer

         ESCRIBIR Contador

      Fin_Para
FIN
```

Otro ejemplo similar al anterior, pero esta vez solicitando a un usuario que ingrese valores, para luego sumarlos:

```
INICIO
Variable i tipo numero <-- 0

Variable suma tipo numero <-- 0

Variable num tipo numero <-- 0

   ESCRIBIR ("Ingrese 10 número que desea sumar")

   Para i <-- 1 hasta 10 hacer

      ESCRIBIR ("Ingrese un número: ")

      LEER num

         suma <-- suma+num
         //en este caso estamos utilizando un acumulador
      Fin Para
      ESCRIBIR ("El resultado es", suma)
FIN
```

En la **Tabla 10** vemos cómo funcionaría internamente este algoritmo repetitivo. Es importante tener en cuenta que la columna de ejemplo de ingreso de número sería lo que un usuario escribiría.

ALGORITMO

NÚMERO DE VECES – VALOR DE I	EJEMPLO DE INGRESO DE NÚMERO	NUM	SUMA+NUM	SUMA
1	5	5	0+5	5
2	2	2	5+2	7
3	7	7	7+7	14
4	1	1	14+1	15
5	8	8	15+8	23
6	5	5	23+5	28
7	3	3	28+3	31
8	6	6	31+6	37
9	4	4	37+4	41
10	1	1	41+1	42

Tabla 10. Funcionamiento del algoritmo. Al finalizar este ciclo PARA, el resultado final que se mostrará en pantalla será el valor 42.

PRUEBA DE ESCRITORIO

La prueba de escritorio es la comprobación lógica de un algoritmo de resolución y constituye una herramienta útil para entender qué hace un determinado algoritmo o verificar que este cumpla con la especificación dada. Para desarrollarla, se requiere el siguiente procedimiento: con datos de prueba, se seguirá cada uno de los pasos propuestos en el algoritmo de resolución. Si la prueba genera resultados óptimos, es que posee una lógica adecuada; en caso contrario, tendrá que ser corregida.

- **Ciclos con un número indeterminado de iteraciones**

A medida que programemos diferentes algoritmos para resolver situaciones, necesitaremos utilizar estructuras que repitan un número de iteraciones que no se conoce con exactitud, ya que depende de un dato dentro del programa.

Mientras Que: esta es una estructura que repetirá un proceso durante "N" veces, siendo "N" fijo o variable. Para hacerlo, la instrucción se vale de una condición que es la que debe cumplirse para que se siga ejecutando; cuando la condición ya no se cumple, el proceso deja de ejecutarse. La sintaxis de esta estructura es la siguiente:

```
Mientras Que <condición> hacer
    Instrucción/es 1 – Acción 1
    Instrucción/es N – Acción N
Fin_Mientras
```

Veamos un ejemplo utilizando la estructura Mientras, aplicando un algoritmo que escriba los números de uno en uno hasta 20:

```
INICIO

Variable contador tipo numero
    contador <-- 1
        Mientras Que contador < 21 hacer
            ESCRIBIR contador
            contador <-- contador + 1
        Fin_Mientras
FIN
```

Repetir-Hasta: esta estructura tiene características similares a la anterior, al repetir el proceso una cierta cantidad de veces; pero a diferencia de **Mientras Que**, lo hace hasta que la condición se cumpla y no mientras se cumple. Por otra parte, esta estructura permite realizar el proceso cuando **menos una vez**, ya que la condición se evalúa al final; en tanto que, con **Mientras Que**, puede ser que nunca llegue a entrar si la condición no se cumple desde un principio. La forma de esta estructura es la siguiente:

```
Repetir
    Instrucción/es 1 – Acción 1
    Instrucción/es N – Acción N
Hasta que <condición>
```

Veamos un ejemplo utilizando la estructura Repetir, en donde realizamos un algoritmo que le pregunta al usuario un número comprendido en el rango de 1 a 5. El algoritmo debe validar el número de manera que no continúe la ejecución del programa hasta que no se escriba un valor correcto:

```
INICIO

Variable num tipo numero
    ESCRIBIR "Escriba un numero de 1 a 5"
        Repetir
            LEER num
            Instrucción/es N – Acción N
        Hasta que (num >= 1) Y (num < 5)
FIN
```


Tipos de datos estructurados

Anteriormente utilizamos datos simples, que representaban un número, un carácter o una cadena/texto. No obstante, a veces necesitamos procesar una colección de valores que estén relacionados entre sí por algún método; por ejemplo: una lista de precios, los meses del año, cotizaciones a lo largo de una semana, etc.

El procesamiento de estos datos utilizando otros simples es muy difícil, porque deberíamos crear, por ejemplo, una variable para cada valor. Por eso, se han definido en la programación varias estructuras de datos que son una colección caracterizada por alguna organización y por las operaciones que se definen en ella. La primera estructura que veremos es el vector.

Vector

Un vector es un conjunto de elementos del mismo tipo de dato que comparten un nombre común; sería una variable que puede almacenar más de un valor al mismo tiempo. Se trata de un **conjunto ordenado** por elementos de posición (de 0 a n) y **homogéneo**, porque sus elementos son todos del mismo tipo de dato. Los vectores también reciben el nombre de **tablas**, **listas** o **arrays**, ya que gráficamente se representa como una tabla.

Un vector de tipo numérico con una dimensión de 5 espacios es:

5	20	-1	10	36

Un vector de tipo alfanumérico con una dimensión de 5 espacios es:

un valor	ABCD	@unvalor	Unvalor200	Texto

De igual forma que cualquier variable, un vector debe tener un nombre:

vecA	5	20	-1	10	36
vecB	un valor	ABCD	@unvalor	Unvalor200	Texto

Los elementos que están en el vector A y en el B ocupan una determinada posición:

vecA	1	2	3	4	5
	5	20	-1	10	36
vecB	1	2	3	4	5
	un valor	ABCD	@unvalor	Unvalor200	Texto

De esta forma, podemos saber la ubicación de los elementos dentro del vector. Por ejemplo, si deseamos saber en el vector A el valor 20, y en el vector B "Unvalor200", hacemos lo siguiente:

```
MOSTRAR vecA(2) //nos mostrará el valor 20
MOSTRAR vecB(2) //nos mostrará el valor Unvalor200
```

Al ver cómo funciona el vector, necesitamos conocer cómo se realiza la declaración de vectores en pseudocódigo. La sintaxis es la siguiente:

```
Variable nombreVector (dimensión vector) tipo dato
```

Como podemos observar, la declaración es igual que la de una variable, pero debemos tener cuidado de ingresar la dimensión del vector entre paréntesis " ( ) " y el tipo de dato que será este vector. Al ser una variable, es un tipo de dato creado en memoria y es **temporal**.

A continuación, veamos un ejemplo de cómo crear y cargar un vector con los datos anteriormente vistos en **vecA**. Por ejemplo, en este caso el pseudocódigo será el siguiente:

```
INICIO
Variable vecA (5) tipo numero
    vecA(1)5
    vecA(2)←20
    vecA(3) 1
    vecA(4)←10
    vecA(5)←36
FIN
```

DIMENSIONAR

No olvidemos que el vector siempre debe ser dimensionado. Esto significa, indicarle a la computadora que reserve los espacios de memoria necesarios para los elementos del vector. En algunos lenguajes, si dejamos el vector sin valores, no podremos hacer uso de él, y eso traerá serios inconvenientes. Por ejemplo: **Variable** vecX () **tipo numero**.

La asignación de valores a los elementos de un vector se realiza indicando el espacio de orden con el signo <- para asignar el valor, de igual manera que con las variables simples. Es posible realizar la carga de vectores con datos predeterminados en código con estructuras de control repetitivas, como **Mientras** o **Hacer-Hasta**.

Por ejemplo, si deseamos realizar la carga de 30 valores a un vector, el pseudocódigo será el siguiente:

```
INICIO
   Variable vecEjemplo (30) tipo numero
   Variable i tipo numero 0

   Para i 1 hasta 30 hacer

      vecEjemplo(i) i

   Fin Para

   MOSTRAR "Vector cargado"
FIN
```

En este ejemplo, la variable i sería "indicador" del espacio en el vector y, además, el valor por asignar. En el ejemplo que se presenta a continuación, dejaremos que el usuario determine la dimensión del vector sobre el que quiere trabajar. Por eso, tomaremos un valor de dimensión y se lo asignaremos al vector:

```
INICIO
   Variable i, num tipo numero <-- 0
   ESCRIBIR "Ingrese la cantidad de valores: ", num

   Variable vecEjemplo (num) tipo numero

   Para i <-- 1 hasta num hacer

      vecEjemplo(i) <-- i
```

```
    Fin Para

    MOSTRAR "Vector cargado"
FIN
```

Hagamos un ejercicio: debemos leer un vector de N componentes, y hallar la suma y el promedio de sus elementos.

Entonces, se pide la suma y el promedio de los elementos. Sabemos que el promedio se encuentra dividiendo la suma de todos los elementos, por la cantidad.

Llamamos a nuestro vector **vecCalculo**, y tendrá una dimensión que será determinada por el usuario. Siguiendo el esquema que vimos con anterioridad, tendremos, primeramente: una repetitiva para la carga del vector, otra para el proceso y otra para mostrar los datos del vector.

```
INICIO
    Variable i, suma, promedio, dimensión, numero tipo numero <-- 0
    ESCRIBIR "Escriba la cantidad de valores a calcular: ", dimensión

    Variable vecCalculo(dimensión) tipo numero

Hacer i <-- 1 hasta dimensión
        ESCRIBIR "Ingrese un número: ", numero
        vecCalculo(i) <-- numero
    Fin Hacer
    //con esta estructura cargamos el vector de valores.

Hacer i <-- 1 hasta dimensión
        suma←suma + vecCalculo(i)
    Fin Hacer
    //con esta estructura sumamos todos los valores del vector.
    promedio <-- suma / 2

        MOSTRAR "La suma de los elementos del vector es: ", suma
        MOSTRAR "El promedio es: ", promedio
FIN
```

Hasta aquí hemos visto: el uso de un vector en pseudocódigo, el beneficio que nos ofrece su estructura para trabajar o registrar varios datos y las distintas operaciones que nos permite realizar.

Matriz

Las matrices son estructuras que contienen datos homogéneos, es decir, del mismo tipo. Así como antes utilizamos un indicador o índice para posicionarnos y almacenar algún valor, en el caso de las matrices, utilizaremos dos índices que determinarán la posición de **fila** y **columna**.

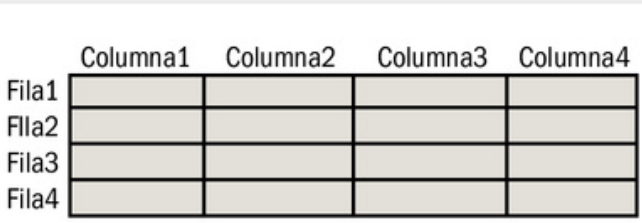


Figura 14. En esta representación tenemos una matriz de dimensión M * N, en donde M es el número de columnas, y N, el número de filas.

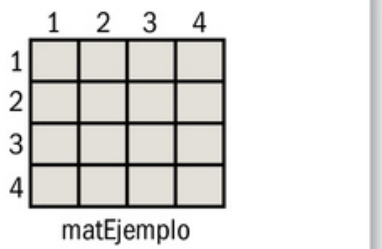


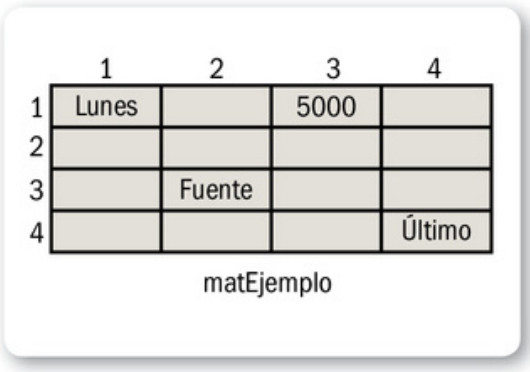
Figura 15. En esta representación de matriz, la sintaxis sería: **`Variable matEjemplo (4, 4) tipo texto.`**

En el ejemplo anterior, la dimensión es M=4 y N=4. Por lo tanto, el número total de elementos es 4*4; es decir, 16 posiciones para utilizar. Al igual que los vectores, una matriz debe tener un nombre. La sintaxis es la siguiente:

Variable nombreMatriz (cantidad filas, cantidad columnas) tipo dato

Una vez que le asignamos datos a la matriz, notaremos que, para referirnos a alguno de sus elementos, tendremos que conocer, precisamente, en qué fila y columna reside este.

Además de cargar los valores de manera independiente, debemos tener en cuenta, al igual que los vectores que utilizaremos, las estructuras repetitivas para recorrer las matrices. Por ejemplo:



	1	2	3	4
1	Lunes		5000	
2				
3		Fuente		
4				Último

matEjemplo

Figura 16. En esta imagen podemos ver la representación de datos correspondientes a la matriz

```
INICIO
   Variable ifila, icolumna tipo numero <-- 0

   Variable varPalabra tipo texto

   Variable matEjemplo (4, 4) tipo texto

   Para ifila <-- 1 hasta 4 hacer

      Para icolumna <-- 1 hasta 4 hacer

         ESCRIBIR "Ingrese un valor: "
         matEjemplo(ifila, icolumna) <-- varPalabra
      Fin Para

   Fin Para

   MOSTRAR "Matriz cargada"
FIN
```

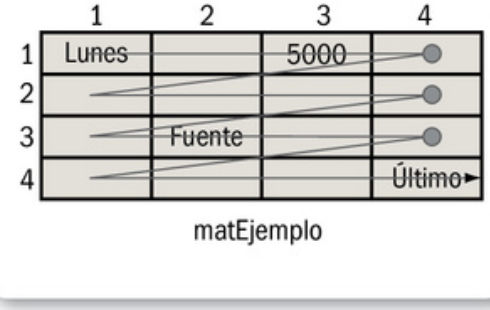


Figura 17. En el recorrido de esta matriz podemos notar los valores que van tomando los índices.

En este ejemplo, la variable fila comienza en el valor 1, luego se da inicio a la repetitiva con la columna desde 1 hasta 4. El bucle de las columnas siempre debe terminar todo su recorrido para que comience el siguiente valor de fila.

Una matriz también puede recorrerse por columnas. Al programar, no siempre podremos predefinir el tamaño de la matriz. Es por eso que necesitamos solicitarle al usuario que ingrese la cantidad de filas y columnas con las que quiere dimensionarla. Por ejemplo:

```
INICIO
   Variable ifila, icolumna tipo numero <-- 0
   Variable varPalabra tipo texto

   ESCRIBIR "Ingrese la cantidad de filas: ", ifila
   ESCRIBIR "Ingrese la cantidad de columnas: ", icolumna

   Variable matEjemplo (ifila, icolumna) tipo texto

   Para icolumna <-- 1 hasta icolumna hacer

      Para ifila <-- 1 hasta ifila hacer
         ESCRIBIR "Ingrese un valor: "
         matEjemplo(ifila, icolumna) <-- varPalabra
      Fin Para

   Fin Para

   MOSTRAR "Matriz cargada"
FIN
```

De esta manera, podemos experimentar cómo cargar una matriz y observar que el recorrido es muy similar al de los vectores. Sin embargo, hay que tener en cuenta que debemos indicar la fila y la columna de la posición de los elementos.

A continuación, veamos un ejemplo:

```
INICIO
   Variable ifila, icolumna tipo numero <-- 0

   Variable matEjemplo (4, 4) tipo texto

   Para ifila 1 hasta 4 hacer

      Para icolumna <-- 1 hasta 4 hacer

         MOSTRAR "El valor es: ", matEjemplo(ifila, icolumna)
      Fin Para

   Fin Para

FIN
```

Para seguir avanzando en el manejo de matrices, a continuación veremos un caso en donde calcularemos los valores entre matrices.

Por ejemplo: supongamos que debemos hacer una suma entre matrices, siendo **matA** y **matB** dos matrices de igual dimensión (**MxN**):

RESOLVER PROBLEMAS

La resolución de un problema mediante una computadora consiste en el proceso que, a partir de la descripción de un problema, expresado habitualmente en lenguaje natural y en términos propios de su dominio, permite desarrollar un programa que lo resuelva. En nuestro caso, se trata de crear el algoritmo para crear el programa que resolverá la situación. Este proceso exige: analizar el problema, diseñar el algoritmo y validación del programa.

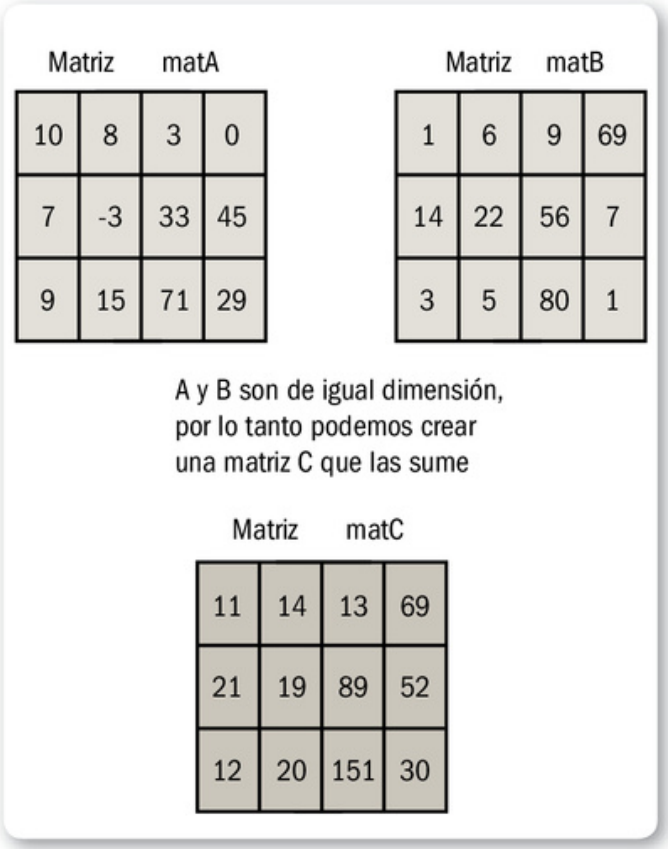


Figura 18. Como matA y matB tienen dimensión MxN, podemos sumarlas y tener como resultado una nueva matriz llamada matC, que conserve la misma dimensión.

```
INICIO
   Variable ifila, icolumna tipo numero <-- 0

   Variable matA (3, 4) tipo numero
   Variable matB (3, 4) tipo numero

   Variable matC (3, 4) tipo numero

   //carga de matriz A
   matA(1,1)←10
   matA(1,2)<-- 8
   matA(1,3)←3
   matA(1,4)←0

   matA(2,1)<-- 7
   matA(2,2)<-- 3
```

```
            matA(2,3)←33
            matA(2,4)<-- 45

            matA(3,1)<-- 9
            matA(3,2)←15
            matA(3,3)<-- 71
            matA(3,4)←29

        //carga de matriz B
            matB(1,1)←1
            matB(1,2)<-- 6
            matB(1,3)<-- 9
            matB(1,4)<-- 69

            matB(2,1)←14
            matB(2,2)←22
            matB(2,3)<-- 56
            matB(2,4)<-- 7

            matB(3,1)←3
            matB(3,2)<-- 5
            matB(3,3)<-- 80
        matB(3,4)←1

//cálculo y asignación a matriz C

        Para ifila <-- 1 hasta 3 hacer

                    Para icolumna <-- 1 hasta 4 hacer

matC(ifila, icolumna) <-- matA(ifila, icolumna) + matB(ifila, icolumna)

                    Fin Para

        Fin Para

FIN
```

Hasta aquí hemos visto el uso de las estructuras complejas que se almacenan en la memoria, los vectores y matrices, que serán útiles para guardar datos de manera temporal. Con toda esta información, podremos realizar el procesamiento de datos, ya sea por cálculo o manejo de texto. A continuación, aprenderemos a automatizar algunas acciones comunes que podemos utilizar en nuestro desarrollo.

Utilizar funciones y procedimientos

Cuando comencemos a practicar el desarrollo de aplicaciones, en nuestro algoritmo habrá cálculos o rutinas que pueden repetirse varias veces. En los próximos párrafos aprenderemos a simplificar la repetición de estos procesos.

PODEMOS RESOLVER UN PROBLEMA COMPLEJO POR MEDIO DE MÓDULOS PEQUEÑOS

En general, un problema complejo puede resolverse de manera eficiente si se divide en procesos pequeños. Esto implica que el problema original será resuelto por medio de varios módulos, cada uno de los cuales se encargará de solucionar alguna parte determinada. Esos módulos se conocen con el nombre de **subalgoritmos**, es decir, algoritmos cuya función es resolver un subproblema. Los subalgoritmos se escriben solo una vez y, luego, podemos hacer referencia a ellos desde diferentes puntos de un pseudocódigo. De esta forma, podemos reutilizar el código y evitar la duplicación de procesos.

Es importante tener en cuenta que los subalgoritmos son independientes entre sí; esto quiere decir que se pueden escribir y verificar en forma separada. Por eso, será más fácil localizar un error en la codificación que estamos creando y, también, modificarlo, sin tener que rehacer varias partes de él. Existen dos clases de subalgoritmos: **funciones** y **procedimientos**, que también encontraremos con los nombres de subrutinas o subprogramas.

Al utilizar procedimientos y funciones veremos que se establece un **límite** para el alcance de las variables; algunas tendrán efecto y valor

solo en el subalgoritmo, y otras, en el algoritmo principal. También es posible especificar que una variable tenga efecto en el algoritmo principal y en todos los subalgoritmos. Este tema se conoce como ámbito de las variables, que pueden ser: locales, privadas o públicas.

Los subalgoritmos pueden recibir valores del algoritmo principal, llamados **parámetros**, trabajar con ellos y devolverle un resultado. También pueden llamar a otros o a sus propios subprogramas; incluso, puede llamarse a sí mismo, lo que se conoce como recursividad.

Ámbito de las variables

En programación, existen dos tipos de variables, las locales y las globales. Las primeras son aquellas que se encuentran dentro de un subprograma, ya sea un procedimiento o una función, y son distintas de las que están en el algoritmo principal. El valor se confina al subprograma en el que está declarada. En cambio, las globales son las que se definen o están declaradas en el algoritmo principal, y tienen efecto tanto en él como en cualquiera de sus subprogramas.

Funciones

Desde el punto de vista matemático, una función es una expresión que toma uno o más valores llamados argumentos y produce un resultado único. Algunos ejemplos de funciones matemáticas son: los logaritmos y las funciones trigonométricas (seno, coseno, etc.).

En el ambiente de la programación de algoritmos, las funciones tienen exactamente el mismo significado. Se realizan ciertos cálculos con una o más variables de entrada, y se produce un único resultado, que podrá ser un valor numérico, alfanumérico o lógico. Es decir, una función puede devolver como resultado una cadena, un número o un valor de tipo lógico. Esto hace que en los lenguajes de programación debamos especificar el tipo de la función.

La función será de tipo **numérica** cuando devuelva un número, y será **alfanumérica** cuando devuelva una cadena. En el caso de las funciones numéricas, existen subdivisiones que están dadas por los tipos de datos soportados por algún lenguaje.

Veamos un ejemplo de la función matemática **sen(x)**. En este caso, la función se llama **sen (seno)**, y el argumento o valor que se le pasa

para que lo procese es **x**. Así, **sen(90°)=1**. Este valor, como es único, se denomina función; es decir, no existe ningún otro número que la función pueda procesar y devolver 1, más que 90°.

Cuando utilicemos esta función en un pseudocódigo y necesitemos el valor del sen(90°), debemos asignarlo de la siguiente forma:

```
variable valor tipo numero 0
valor <-- sen(90)
```

Aquí, como la variable valor es 1, nuestra función es numérica. Es así como se llama a las funciones desde un pseudocódigo, asignándolas siempre a una variable que contendrá el valor devuelto por la función. Si no hacemos esta asignación, la función no podrá ejecutarse, porque no tendrá un espacio o lugar donde descargar el resultado. Por lo tanto, la llamada a una función tendrá la siguiente sintaxis:

```
variable <-- funcion (parámetros)
```

Veamos un ejemplo: si a la función **MES**, que devuelve el nombre del mes, le pasamos el valor numérico correspondiente, el resultado será:

```
variable nombre_mes tipo texto
nombre_mes <-- MES(2) //esto devolvería "Febrero"
```

La función es de tipo texto, porque devuelve una cadena como resultado en la variable **nombre_mes**.

Hasta aquí hemos visto cómo llamar a una función, ahora veremos cómo escribirla. Las funciones y los procedimientos no se escriben en el algoritmo principal, ya que, en programación, existen espacios destinados a ellos. Todas las funciones y los procedimientos que utilicen un algoritmo podrán escribirse antes o después del algoritmo principal.

Una función se identifica por su nombre, como cuando escribimos un algoritmo utilizando inicio y fin para indicar dónde comienza y dónde termina. A continuación, veamos cómo sería la sintaxis:

```
Función nombre_funcion (parámetros)
 Instrucción/es
Fin función
```

Todas las funciones devuelven **un solo valor**. Siempre debemos indicar a la función, mediante una instrucción, que devuelva el valor al algoritmo principal; recordemos que la función será llamada desde un algoritmo. Esto se debe hacer en el cuerpo de la función cuando tengamos el resultado. Tomando como ejemplo la función **MES**, veremos cómo se escribe el algoritmo principal, cómo se llama y se declara la función:

```
INICIO
        variable numero_mes tipo numero
        variable nombre_mes tipo texto

        ESCRIBIR "Ingrese el número del mes y le mostraremos el nombre del mes"

           ESCRIBIR "Debe ingresar un número entre 1 y 12: " , numero_mes

        Si numero_mes>12 o numero_mes<1 entonces

MOSTRAR "Debe ingresar un valor entre 1 y 12"
        Sino
                  nombre_mes  <-- llamar MES(numero_mes)

                  MOSTRAR "El mes correspondiente es: ", nombre_mes
        FinSi
FIN
```

```
FUNCION MES (variable valor tipo numero)
variable nombre tipo texto

   Según sea valor
      Caso 1:
nombre="Enero"
      caso 2:
nombre= "Febrero"
      caso 3:
nombre = "Marzo"
      caso 4:
```

```
nombre = "Abril"
      caso 5:
nombre = "Mayo"
       caso 6:
nombre = "Junio"
       caso 7:
nombre ="Julio"
       caso 8:
nombre="Agosto"
       caso 9:
nombre="Setiembre"
       caso 10:
nombre = "Octubre"
       caso 11:
nombre= "Noviembre"
       caso 12:
nombre="Diciembre"
   Fin Según
```

MES <-- nombre //Indicamos a la función que devuelva el resultado al algoritmo principal la variable nombre

FIN FUNCION

Es preciso tener en cuenta cómo se pasan los valores desde el algoritmo principal a la función. En este caso, cuando se llama a la función, escribimos:

nombre_mes <-- MES (numero_mes)

El valor que se envía a la función **MES** de la variable **numero_mes** toma un valor comprendido entre 1 y 12. Cuando se llama a la función, este valor debe ser recibido por ella; en este caso, en el cuerpo de la función se coloca entre paréntesis el nombre de la variable que recibirá el valor:

Función MES (valor)

Si se pasan varios valores, todos deben ser recibidos en sus correspondientes variables, o habrá un error en la codificación. La función toma el valor pasado desde el algoritmo y lo guarda en la variable **valor** para procesarlo. Luego de que obtiene un resultado, en este caso el valor de **nombre_mes**, se le ordena a la función que devuelva ese valor al algoritmo principal:

MES <-- nombre
Sintaxis: nombre_funcion <-- resultado

En la mayoría de los lenguajes de programación se utiliza una palabra reservada para devolver valores: **return**. En el caso de pseudocódigo, también podemos usar **devolver**.

En resumen, podemos decir que la función devuelve **un solo valor**, que para funcionar debe recibir uno o varios valores desde el algoritmo principal, realizar el proceso y devolver el resultado. La función se escribe de igual manera que cualquier algoritmo; la diferencia consiste en que, en vez de inicio y fin, escribimos:

```
Función nombre_funcion (parámetros)
 Instrucción/es
Fin función
```

Veamos otro ejemplo: diseñar el algoritmo para realizar la raíz cuadrada de un valor numérico. En este caso, el algoritmo es el siguiente:

```
INICIO
      variable num, resultado tipo numero <-- 0
     ESCRIBIR "Ingrese un número:", num

     Mientras num < = 0
              MOSTRAR "Ingrese un número positivo"
              ESCRIBIR "Ingrese un número:", num
      Fin Mientras

     resultado <-- RAIZ(num)
    MOSTRAR "La raíz cuadrada es:", resultado
```

```
FUNCION RAIZ (variable valor tipo numero)
      variable varRaiz tipo numero
      varRaiz <-- valor ^ 1/2
      RAIZ <--  varRaiz
FIN FUNCION
```

Llamamos **RAIZ** a la función que escribimos, la cual debe obtener un valor que se pasará desde el algoritmo principal. El número del cual queremos calcular la raíz cuadrada lo elevamos a la potencia 1/2 y, luego, devolvemos el resultado al algoritmo principal.

Tomando como referencia esta función **RAIZ**, las variables globales son **num** y **resultado**, y las locales son: **valor** y **varRaiz**. Estas dos últimas solo existen en la función **RAIZ**, y si en el algoritmo principal tratamos de utilizar estas variables o mostrarlas, no obtendremos nada, ya que para el algoritmo ellas son locales y no existen.

Las variables **num** y **resultado** son globales, es decir que están disponibles en el algoritmo principal y, también, en la función **RAIZ**.

Una variable local de un subprograma no tiene ningún significado en el algoritmo principal y en otros subprogramas. Si un subprograma asigna un valor a una de sus variables locales, este no será accesible a otros subprogramas, es decir que no podrán usar este valor. Las variables globales tienen la ventaja de compartir información de diferentes subprogramas.

Para terminar con el tema de funciones, veremos un ejemplo para diseñar una función que calcule la media de tres números:

```
INICIO
       variable numero1, numero2, numero3 tipo numero ←0
       variable prom tipo numero ←0

      ESCRIBIR "Ingrese tres valores: ", numero1, numero2, numero3

      prom <-- PROMEDIO(numero1, numero2, numero3)

      MOSTRAR "El promedio es:", prom
FIN
```

```
Funcion PROMEDIO(variable valor1, valor2,valor3 tipo numero)
    variable promedio tipo numero <-- 0

    promedio (valor1 + valor2 + valor3) / 3

    PROMEDIO <-- promedio
Fin Función
```

Hasta aquí hemos visto que las funciones se utilizan para devolver como resultado un valor. En ocasiones, necesitaremos devolver más de un resultado o, también, ejecutar las mismas líneas de código varias veces en un algoritmo, como una ordenación. En estas situaciones la función no es apropiada, y utilizaremos los procedimientos, también llamados subrutinas.

Procedimientos

Un procedimiento es un conjunto de sentencias o instrucciones que realizan una determinada tarea y que pueden ser ejecutados desde más de un punto del programa principal. Este tiene una llamada y, cuando se ejecuta totalmente, vuelve al punto desde donde fue llamado y se ejecuta la siguiente instrucción.

El procedimiento se escribe como cualquier otro algoritmo, ya que solo existen diferencias en la parte inicial y final. Para nombrar los procedimientos, hay que seguir las mismas reglas que para las variables. El objetivo de los procedimientos es ayudar en la modularidad del programa y evitar la repetición de instrucciones, porque estas pueden escribirse en un procedimiento y, en lugar de repetirlas, podemos llamar al procedimiento cuantas veces sean necesarias.

Desde el programa principal, es posible pasar valores al procedimiento, que los utilizará para realizar un determinado proceso. Los valores se llaman **parámetros**, y la sintaxis para la declaración de un procedimiento es la siguiente:

```
Procedimiento Nombre_procedimiento (parámetros)
    Instrucción/es
Fin Procedimiento
```

La llamada a un procedimiento se hace por su nombre:

Nombre_procedimiento(parámetros)

También es posible que no se pase ningún parámetro al procedimiento, en cuyo caso la llamada se escribe de la siguiente manera:

Nombre_procedimiento()

Cuando no se pasan parámetros, se pueden obviar los paréntesis, pero es una buena forma de escritura algorítmica escribirlos, por ejemplo:

Nombre_procedimiento

Podemos utilizar procedimientos, por ejemplo, para: dibujar recuadros en la pantalla, mostrar mensajes de error, realizar procesos con más de un resultado y colocar en un procedimiento las líneas de código que se repiten varias veces en un algoritmo.

Cuando necesitamos devolver más de un valor en un procedimiento, las variables que devuelvan los resultados deben figurar en la lista de parámetros.

Veamos un ejemplo del procedimiento para calcular el cociente y resto de la división entre dos números:

```
INICIO
      variable numeroA, numeroB tipo numero ←0

      ESCRIBIR "Ingrese los valores a calcular: ", numeroA, numeroB
     DIVISION (numeroA, numeroB, P, Q)

     MOSTRAR P, Q
FIN
```

```
Procedimiento DIVISION (variable dividendo, divisor, cociente, resto tipo numero)

   cociente  <-- dividendo / divisor
```

```
    resto  <-- dividendo - cociente * resto

Fin Procedimiento
```

Al llamar al procedimiento división en el algoritmo principal, debemos pasar en su sintaxis los números del dividendo y del divisor, que están representados por **numeroA** y **numeroB**. También hay que especificar las variables en las que se devolverán los resultados del cociente y el resto, que serán P y Q. De esta forma, la sintaxis de la llamada quedará así:

DIVISION (numeroA, numeroB, P, Q)

El procedimiento recibe los valores: **numeroA** en dividendo, **numeroB** en divisor, y se colocan las variables en las que se pasarán al programa principal el cociente y el resto. De esta forma, P recibirá el valor de cociente, y Q, el del resto.

Cuando necesitamos devolver más de un valor, los parámetros del procedimiento deben ser los valores que se pasen al procedimiento y, luego, las variables en las que se recibirán los resultados.

RESUMEN

Hemos aprendido a crear algoritmos por medio de pseudocódigo, el lenguaje "común" que podemos utilizar para cualquier lenguaje de programación. Recorrimos la creación de algoritmos, sus variables, estructuras de control, tipos de datos, funciones y procedimientos. Todas estas herramientas nos permiten trabajar con cierta lógica para resolver situaciones específicas y brindar resultados. De esta manera, lograremos darle vida al funcionamiento de programas que permitan resolver situaciones particulares.

Actividades

TEST DE AUTOEVALUACIÓN

1. ¿Cuál es la forma de razonamiento de un humano y de una computadora?
2. ¿Cómo podemos expresarnos para mostrarle a otra persona el funcionamiento interno de un programa?
3. ¿Qué normas debemos tener en cuenta en el armado de pseudocódigo?
4. ¿Qué son los tipos de datos? ¿Cuáles son los más usuales para los ejemplos de pseudocódigo?
5. ¿Cuál es la diferencia entre contador, acumulador y variable?
6. ¿Para qué se usan los operadores aritméticos?
7. ¿Cuál es la ventaja de utilizar operadores lógicos?
8. ¿Cómo se emplea la asignación en pseudocódigo?
9. En pseudocódigo, ¿cuáles son las palabras que podemos utilizar para indicar a los usuarios que escriban algo y, luego, capturarlo?
10. ¿Cómo se usan las estructuras de control simples, dobles y anidadas?

ACTIVIDADES PRÁCTICAS

1. Haga un pseudocódigo que imprima todos los números naturales que hay, desde la unidad hasta un valor indicado por teclado. Luego, introduzca las frases que desee y cuéntelas.
2. Haga un pseudocódigo que permita introducir solo S o N. Luego imprima y cuente los múltiplos de 3, desde la unidad hasta un número introducido por teclado.
3. Imprima diez veces la serie de números del 1 al 10 y haga un pseudocódigo que cuente las veces que aparece una letra en una frase ingresada por teclado.
4. Escriba un pseudocódigo que imprima los números del 0 al 100, controlando las filas y las columnas. Luego simule cien tiradas de dos dados y cuente las veces que entre los dos suman 10.
5. Introduzca dos números por teclado y, mediante un menú, calcule la: suma, resta, multiplicación y división entre ellos.

*04

Primer proyecto en Visual Basic

A lo largo de los capítulos anteriores, recorrimos conceptos, teorías y prácticas para el desarrollo de aplicaciones informáticas. Ahora plasmaremos todo el conocimiento adquirido sobre pseudocódigo en el código fuente de un lenguaje, empezando con las interfaces gráficas y, luego, con el código fuente.

- **Lenguajes de programación 132**
  - Tipos de lenguajes 132
- **Interfaces gráficas 134**
  - Nomenclatura en pseudocódigo y lenguajes de programación 138
- **Lenguaje de programación: Microsoft Visual Basic 140**
  - Creación de proyectos 140
  - Qué son y cómo se usan las variables 146
  - Cómo se utilizan los operadores 154
  - Todo tiene un orden en la programación 159
  - Tipos de datos estructurados 168
  - Uso de controles básicos 175
- **Resumen 187**
- **Actividades 188**

Lenguajes de programación

En este capítulo nos adentraremos en algunos lenguajes de programación y comenzaremos a aplicarles lo aprendido sobre pseudocódigo, para dar así nuestros primeros pasos en un código fuente para el desarrollo de aplicaciones.

Los lenguajes de programación son definidos como un **idioma** artificial diseñado para expresar cómputos que pueden ser llevados a cabo por equipos electrónicos, tales como computadoras, tablets, smartphones, etc. El uso de este **lenguaje máquina**, que vimos en capítulos anteriores, nos permitirá crear programas o aplicaciones que controlen el comportamiento físico y lógico de un dispositivo electrónico (expresado en algoritmos de precisión) y, además, establecer la comunicación humano-máquina. Su escritura está formada por un conjunto de símbolos, reglas sintácticas y semánticas que definen la estructura y el significado de sus elementos y expresiones, al igual que las reglas ortográficas lo hacen con el lenguaje humano. Por último, debemos tener en cuenta el proceso de **programación** por el cual se escribe, prueba, depura, compila y mantiene el código fuente de un programa informático, al que nosotros llamaremos **desarrollo**.

> LA ESTRUCTURA DEFINE EL SIGNIFICADO DE SUS ELEMENTOS Y EXPRESIONES

Tipos de lenguajes

Existen lenguajes sólidos, duros, visuales, amigables y específicos en la programación de código fuente. En la **Tabla 1**, podemos ver la variedad que hay y cuáles son sus diferencias.

LENGUAJES DE BAJO NIVEL

La palabra "bajo" no implica que el lenguaje sea inferior a un lenguaje de alto nivel, sino que se refiere a la reducida abstracción entre **el lenguaje y el hardware**. Por ejemplo, estos lenguajes se utilizan para programar controladores de dispositivos, tales como placas de video, impresoras u otros.

LENGUAJES	
▼ TIPO	▼ DESCRIPCIÓN
Lenguaje máquina	Tanto las invocaciones a memoria como los procesos aritmético-lógicos son posiciones literales de conmutadores físicos del hardware en su representación booleana. Estos lenguajes son literales de tareas.
Lenguaje objeto	Lenguaje o juego de instrucciones codificado al cual es traducido un lenguaje fuente por medio de un compilador. Es un lenguaje máquina directamente comprensible por una computadora.
Lenguajes de bajo nivel	Ensamblan los grupos de conmutadores necesarios para expresar una mínima lógica aritmética, y están íntimamente vinculados al hardware. Estos lenguajes están orientados a procesos, y el diseño de la arquitectura de hardware determinará la cantidad de instrucciones.
Lenguajes de medio nivel	Basándose en los juegos de instrucciones disponibles (chipset), permiten el uso de funciones a nivel aritmético, pero a nivel lógico, dependen de literales en ensamblador. Estos lenguajes están orientados a procedimientos. Ejemplos: C y Basic.
Lenguajes de alto nivel	Le permiten al programador tener una máxima flexibilidad a la hora de abstraerse o ser literal, y ofrecen un camino bidireccional entre el lenguaje máquina y una expresión casi oral entre la escritura del programa y su posterior compilación. Estos lenguajes están orientados a objetos, que a su vez, se componen de propiedades cuya naturaleza emerge de procedimientos. Ejemplos: C++, Fortran, Cobol y Lisp.
Lenguajes de aplicaciones	No permiten una bidireccionalidad conceptual entre el lenguaje máquina y los lenguajes de alto nivel, ni tampoco la literalidad a la hora de invocar conceptos lógicos. Se basan en librerías creadas en lenguajes de alto nivel. Pueden permitir la creación de nuevas librerías, pero propietarias y dependientes de las suministradas por la aplicación. Están orientados a eventos que surgen cuando las propiedades de un objeto interactúan con otro. Ejemplo: Visual Basic para aplicaciones.
Lenguajes de redes	Se basan en un convenio de instrucciones independientes de la máquina y dependientes de la red a la que están orientadas. Se dividen en descriptivos (HTML, XML, VML) y cliente-servidor (Java, PHP) y de script.

Tabla 1. Clasificación y detalle sobre los distintos lenguajes de programación.

Esta tabla nos da un detalle sobre los tipos de lenguajes de programación que podemos encontrar en el mercado informático. Frente a esto, es importante tener en cuenta que, en general, hablamos de lenguajes de **alto nivel** y de **bajo nivel**, agrupando en ellos todos los tipos que vimos antes. Para comprender mejor este punto, podemos generalizar estos conceptos diciendo que el lenguaje de **bajo nivel** es el que más se asemeja al lenguaje máquina (por ejemplo: lenguaje Assembler), en tanto que el de **alto nivel** se asemeja al lenguaje humano y, por medios visuales, nos permite crear nuestros desarrollos (por ejemplo: Visual Basic).

Compilador

Como vimos en capítulos anteriores, un lenguaje utilizado para escribir programas de computación permite la comunicación **usuario-máquina**. A su vez, existen algunos programas especiales llamados **traductores** (**compilador, intérprete**) que convierten las instrucciones escritas en código fuente a un lenguaje máquina que el equipo electrónico pueda comprender y procesar.

Interfaces gráficas

Las interfaces gráficas son aquellas que nos permiten comunicarnos con un dispositivo, y el concepto que podemos encontrar es **comunicación o interacción usuario-máquina**. Gracias a la evolución de las interfaces de comunicación usuario-máquina, podemos apreciar medios gráficos para la interacción con los diferentes

GUI

Graphic User Interface, o interfaz gráfica de usuario, abarca un conjunto de formas que hacen posible la interacción usuario-máquina por medio de elementos gráficos e imágenes. Cuando hablamos de elementos gráficos, nos referimos a botones, iconos, ventanas, tipos de letras, etcétera, que representan funciones, acciones e información.

componentes de un equipo. Por ejemplo: el manejo de impresora, mouse, monitor, etc.

En la confección de interfaces gráficas, debemos tener en cuenta que hay ciertas normas por cumplir, y que los componentes utilizados en una interfaz tienen una nomenclatura específica. Por ejemplo, en la **Tabla 2** se presenta una lista de controles que podemos encontrar en distintos lenguajes de programación.

INTERFAZ GRÁFICA

▼ TIPO	▼ COMPONENTE	
Comandos	• Botón de comando • Menú contextual	• Menú (y submenú) • Menú desplegable
Entada/salida de datos	• Casilla de verificación • Lista • Lista desplegable (combo box) • Botón de opción (radio button)	• Caja de texto (text box) • GridView (datagrid) • Barra de desplazamiento (scrollbar)
Informativos	• Caja de texto (text box o cuadro de texto) • Etiqueta (label) • Icono • Barra de estado (status bar) • Globo de ayuda (ballon help) • Barra de progreso (progress bar) • Barra de título • Slider	• Spinner • HUD (heads-up) • Infobar • Splash screen • Throbber • Toast • Tooltip
Contenedores	• Ventana/Form/Forma • Barra de menú (menu bar) • Pestaña (tab) • Panel • Cuadro (frame/fieldset)	• Barra de herramientas • Acordeón • Ribbon • Disclosure widget (expansor o Combutcon)
De navegación	• Vista de árbol (treeview) • Vista en lista (listview) • Barra de direcciones	• Breadcrumb • Hipervínculo

Ventanas especiales	• Acerca de (about box) • Cuadro de diálogo (dialog box) • Cuadro de diálogo de archivos	• Inspector Window • Modal Window • Ventana de paleta
Gráfica	• Caja de imagen (picture box)	• Imagen (image)

Tabla 2. Estos son los componentes más frecuentes que podemos encontrar dentro de las interfaces gráficas.

Cabe destacar que estos controles sirven para aplicaciones tanto de escritorio como web. En la **Figura 1** vemos los controles más utilizados en las aplicaciones de Visual Basic.

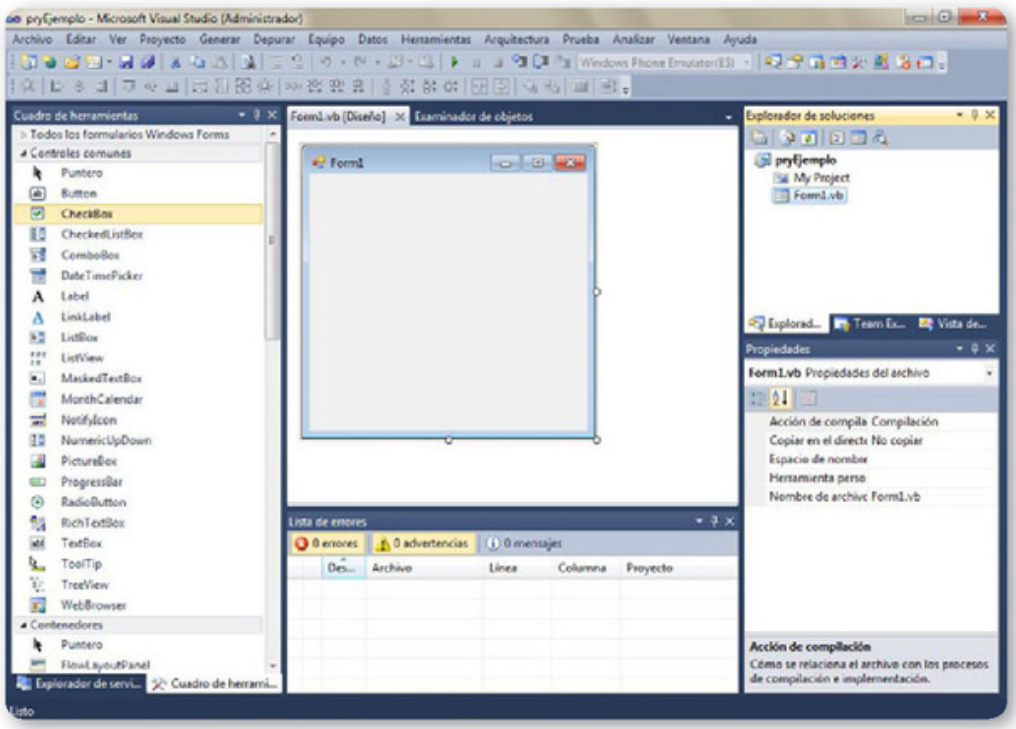


Figura 1. Controles comunes en el IDE de Visual Studio, donde podemos destacar **`Button`**, **`Label`** y **`TextBox`**, entre otros.

Antes de determinar las reglas de diseño de las interfaces gráficas, es importante considerar cuál es el ámbito en el que vamos a trabajar. Esto se debe a que, dependiendo de si es una aplicación de escritorio, móvil o web, las reglas no serán iguales, sino que deberán ajustarse

en función del producto que queramos desarrollar. A continuación, vamos a ver cuáles son las normas generales que podemos utilizar al incursionar en el diseño de interfaces gráficas:

- **Anticipación**: una buena aplicación intentará predecir las necesidades y los deseos de los usuarios.
- **Autonomía**: el usuario debe verse incentivado a investigar y sentir que tiene el control de la interfaz. No obstante, hay quienes se sienten más cómodos en un entorno explorable, que no sea demasiado grande o restrictivo. Para maximizar el rendimiento de un negocio, debemos maximizar la eficacia de todos los usuarios, y no solo de un grupo de ellos.
- **Coherencia**: si queremos que nuestra aplicación sea coherente, es preciso realizar varias pruebas con sus futuros usuarios, para así asegurarnos de que el diseño sea el indicado.
- **Eficiencia**: debemos lograr que la aplicación potencie la experiencia del usuario y le facilite sus tareas, en vez de buscar la potenciación del equipo informático.
- **Aprendizaje**: lo ideal sería que el usuario pudiera sentarse delante del sistema y saber cómo utilizarlo sin necesidad de aprendizaje. Sin embargo, esto casi nunca sucede.
- **Comunicación**: es preciso mantener siempre informado al usuario sobre el estado del sistema, mediante cuadros de diálogo, etiquetas, colores e iconos, y escribiendo mensajes de ayuda concisos que resuelvan los posibles inconvenientes.

Como podemos ver, a lo largo del desarrollo y el diseño de interfaces gráficas podemos encontrar una gran variedad de reglas que debemos tener en cuenta en función de las aplicaciones, el usuario y el tipo de negocio al que nos estemos dirigiendo.

CLI (*COMMAND LINE INTERFACE*)

La interfaz de línea de comando es un método que les permite a las personas dar instrucciones al programa informático por medio de comandos, sin necesitar la intemediación de interfaces gráficas. Por ejemplo, podemos encontrar sistemas operativos como el antiguo DOS y, en la actualidad, consolas de administración en distribuciones de Linux.

Nomenclatura en pseudocódigo y lenguajes de programación

Las nomenclaturas hacen referencia a una lista de nombres de algún tipo de objetos. Así como en química se utiliza una nomenclatura para diferenciar los elementos de la tabla periódica, en programación se la usa para reglamentar abreviaciones que hagan referencia a componentes y, así, saber a qué estamos llamando desde el código fuente; por ejemplo, si estamos haciendo referencia a un botón o a una caja de texto con su nombre particular.

En la **Tabla 3** veremos la nomenclatura que suele emplearse para representar algunos controles comunes de los lenguajes, y también una breve reseña acerca de su aplicación.

NOMENCLATURA

TIPO DE CONTROL	NOMENCLATURA	USO
Botón de comando / Button	Btn	Ejecuta un comando o una acción.
Casilla de verificación / CheckBox	chk	Presenta una opción de tipo Verdadero o Falso.
Lista desplegable / ComboBox	cbo	Combina un cuadro de texto y uno de lista.
Etiqueta / Label	lbl	Presenta texto con el cual el usuario no puede interactuar ni modificar.
Caja de lista / ListBox	lst	Ofrece una lista de elementos entre los que el usuario puede elegir.
Caja de imagen / PictureBox	pic	Presenta mapas de bits, iconos o metarchivos de Windows, y otros tipos de archivos gráficos compatibles. También ofrece texto o actúa como contenedor visual para otros controles.

Botón de opción / RadioButton	opt	Forma parte de un grupo de opciones y presenta varias alternativas entre las que el usuario solo puede elegir una.
Caja de texto / TextBox	txt	Proporciona un área para escribir o presentar texto.
Forma / Form / Formulario / Ventana	Frm	Contenedor de controles.

Tabla 3. Nomenclatura de controles comunes.

Como podemos ver en la tabla anterior, las nomenclaturas que utilizamos en los controles suelen ser abreviaturas de sus nombres originales. En general, utilizando ese nombre que está en inglés, podremos identificar fácilmente a qué control nos estamos refiriendo. Por ejemplo, si el control se llamara **Grilla / GridView**, la posible nomenclatura sería **grd**, en referencia a GRID.

Para evitar confusiones, es bueno consensuar con un equipo de desarrollo la forma que se adoptará para nombrar controles en el código fuente que vamos a desarrollar.

Las interfaces gráficas son elementos fundamentales para poder representar una aplicación adecuada e intuitiva y, así, lograr que el usuario se sienta cómodo al transitarla. En el caso de la confección de interfaces gráficas, debemos conocer los diferentes controles que podemos emplear, además de tener en claro la nomenclatura implicada en cada uno de ellos para el código fuente.

CARACTERÍSTICAS DE LOS CONTROLES

Las **Propiedades** son las características que definen los rasgos de un control, tales como su color, texto, tamaño, etc.; y los **Métodos** son aquellos procesos que puede llevar a cabo un control por sí mismo. En cambio, cuando hablamos de **Eventos**, nos referimos a las acciones que se realizan sobre un control, generalmente, por medio de la interacción del usuario.

Lenguaje de programación: Microsoft Visual Basic

En esta parte vamos a aplicar los conocimientos vistos, plasmándolos en el legendario lenguaje de programación Visual Basic. Utilizaremos el entorno de desarrollo **IDE** (**Integrated Development Environment**) de Visual Studio 2010 versión Visual Basic Express.

Otros tipos de IDE:

- Eric: **http://eric-ide.python-projects.org**
- Mono: **www.monodevelop.com**
- Wingware: **www.wingware.com**
- NetBeans: **http://netbeans.org/community/releases/61**
- Visual Studio: **http://www.microsoft.com/visualstudio**

Para instalar esta versión, podemos hacerlo desde la página oficial de Microsoft: **www.microsoft.com/visualstudio/latam**. También podemos encontrar diferentes videos sobre la instalación, tanto en páginas de Microsoft como en YouTube.

Creación de proyectos

Para comenzar a utilizar el lenguaje de programación, debemos conocer qué es un **proyecto o solución** en Visual Basic. Un proyecto está compuesto por un conjunto de carpetas y archivos que nos permitirán armar una aplicación, en donde se almacenan códigos fuente, librerías, interfaces, etc.

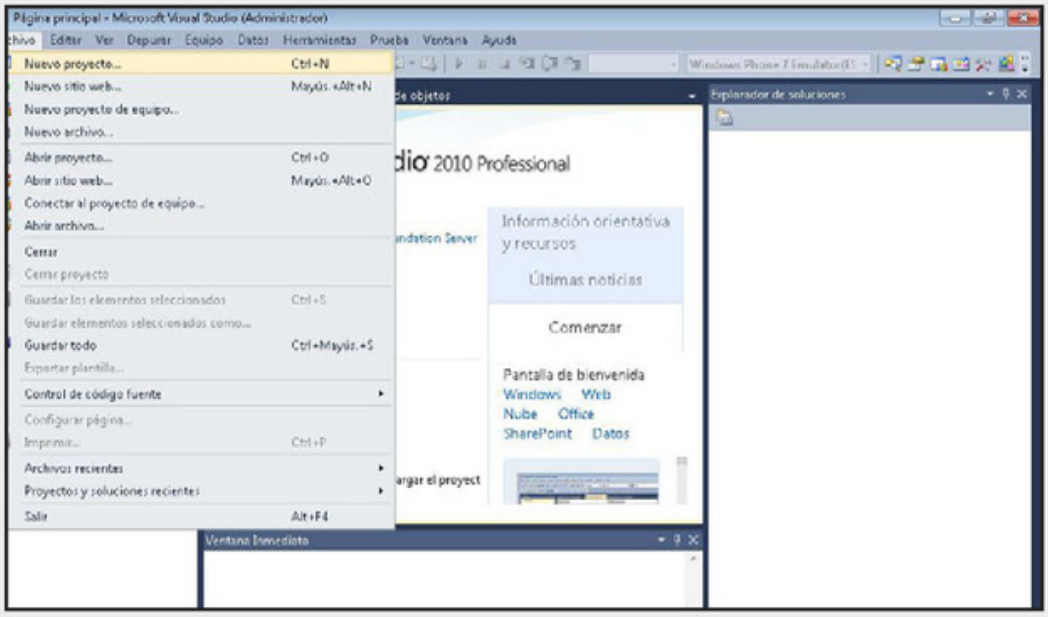
Para crear un proyecto debemos tener en cuenta el siguiente instructivo, que tomaremos como etapa principal para todos los paso a paso que presentaremos a lo largo del capítulo.

NETBEANS

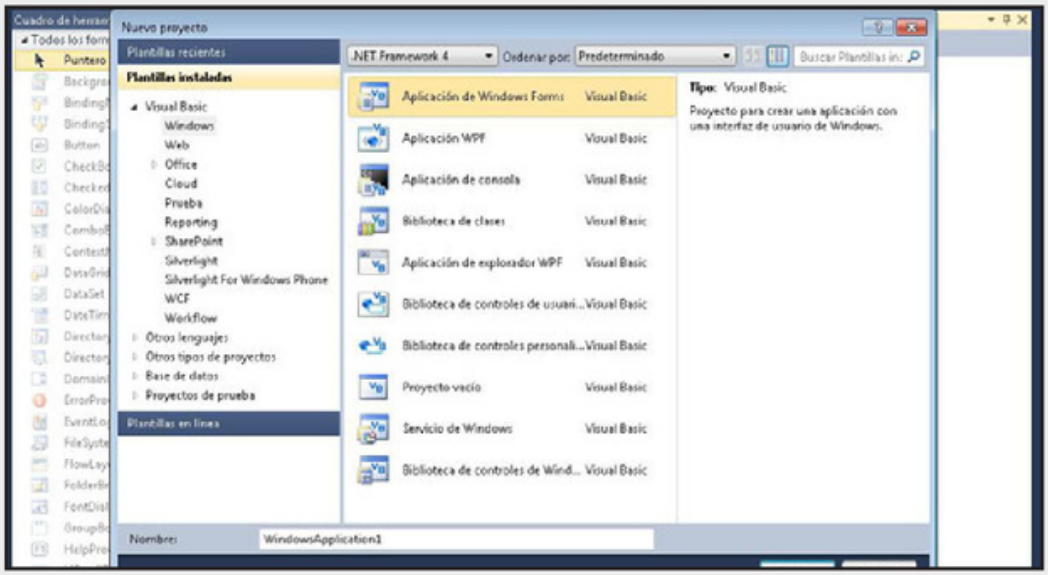
NetBeans es un proyecto de código abierto con una gran base de usuarios, una comunidad en constante crecimiento y cerca de 100 socios en todo el mundo. **Sun MicroSystems** fundó el proyecto de código abierto **NetBeans** en junio de 2000, y hoy continúa siendo su patrocinador principal.

PASO A PASO: CREACIÓN DE UN PROYECTO

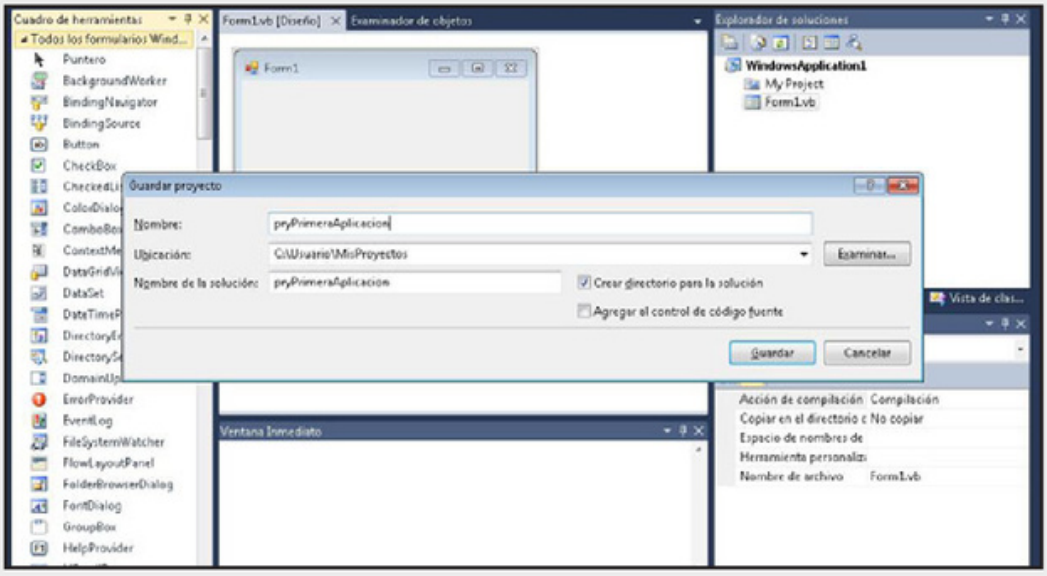
01 Ejecute Visual Studio y, desde el menú **Archivo**, haga clic en **Nuevo proyecto**. También puede hacerlo desde la interfaz principal o utilizando las teclas **CTRL+N**.



02 Dentro de Visual Basic, seleccione **Aplicación de Windows Forms** para crear una aplicación con Windows y haga clic en **Aceptar.**



03 Aparece un nuevo formulario y se agregan los archivos necesarios. Si es el primer proyecto que crea, recibirá el nombre de "WindowsApplication1". Al momento de guardarlo, elija el nombre y haga clic en **Guardar.** La nomenclatura adecuada sería PRY, que proviene de proyecto. Por ejemplo, pryPrimeraAplicacion.



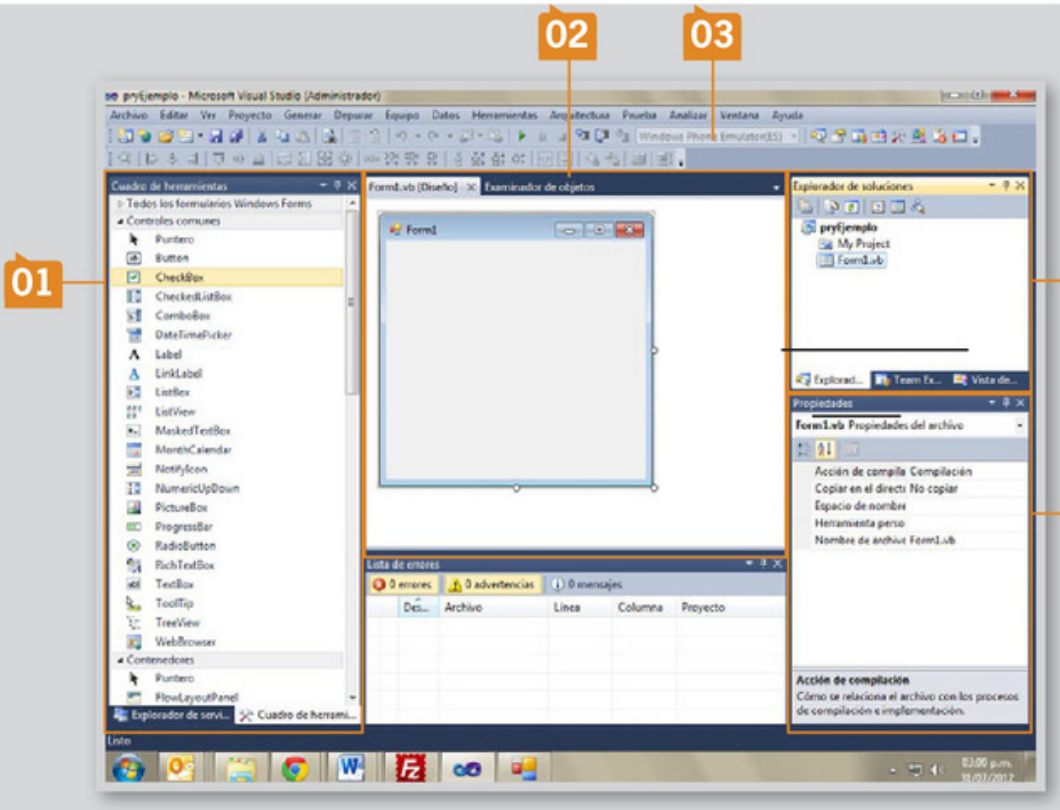
Una vez que hayamos concluido con este paso a paso, obtendremos como resultado nuestro primer proyecto. Desde aquí, podremos ver diferentes herramientas en el IDE que nos permitirán realizar las tareas y administrar el proyecto.

Para reforzar este concepto, a continuación desarrollaremos en detalle las características generales del IDE, representadas en la siguiente guía visual. Aquí podremos ver: el cuadro de herramientas, el diseñador de formularios, el editor de código, el explorador de soluciones y las propiedades.

IDE

Un IDE es un entorno de programación que ha sido empaquetado como un programa de aplicación. Consiste en un **editor de código, compilador, depurador y constructor de interfaz gráfica** (GUI). Los IDEs pueden ser aplicaciones por sí solas o formar parte de aplicaciones existentes. Por ejemplo: Visual Studio, Proyecto Mono, etc.

CARACTERÍSTICAS GENERALES DEL IDE



01 **CUADRO DE HERRAMIENTAS**: proporciona las herramientas disponibles durante el diseño para colocar controles en un formulario. Además del diseño predeterminado, permite crear un diseño personalizado seleccionando **Agregar Ficha** en el menú contextual y añadiendo los controles a la ficha resultante.

02 **SOLAPA DISEÑADOR DE FORMULARIOS**: personaliza el diseño de la interfaz, al permitir el agregado de controles, gráficos e imágenes a un formulario. Cada formulario de la aplicación tiene su propia solapa **Diseñador de formulario**.

03 **VENTANA EDITOR DE CÓDIGO**: funciona como editor para escribir el código de la aplicación. Para cada formulario o clase de código se crea una ventana diferente.

04 **VENTANA EXPLORADOR DE SOLUCIONES**: permite acceder a los formularios, componentes, clases, etc. Desde ella se puede ver el diseño gráfico de dichos formularios (botón **Ver Diseñador**) y editar el código que contienen (botón **Ver Código**). Existen otras funciones, como: **Actualizar** y **Diseñador de clases**, entre otras.

05 **VENTANA PROPIEDADES**: enumera las propiedades del objeto seleccionado y su correspondiente valor. Además, muestra el significado de la propiedad mediante una breve descripción. Permite agrupar las propiedades de acuerdo con un tipo y ordenarlas alfabéticamente. Con el botón **Rayo** podemos visualizar los eventos correspondientes al objeto seleccionado.

Teniendo en cuenta estas características, podremos administrar mejor nuestro desarrollo, ubicando fácilmente las diferentes herramientas y ventanas, tanto al momento de diseñar la interfaz, como en la escritura del código fuente.

ARCHIVOS

EXTENSIÓN DE ARCHIVO	DESCRIPCIÓN
.SLN	Se utiliza para archivos de solución que enlazan uno o más proyectos; almacena información global. Los archivos .SLN son similares a los archivos de grupo Visual Basic (.vb), que aparecen en versiones anteriores de Visual Basic.
.SUO	Se utiliza para archivos que acompañan los registros de solución y las personalizaciones que agreguemos a nuestra solución. Este archivo guarda nuestra configuración, como puntos de interrupción y elementos de tareas, para que sean recuperados cada vez que abramos la solución.
.VBPROJ	Proyecto en Visual Basic.
.CSPROJ	Proyecto en C#.
.VB	Archivo con código Visual Basic.
.CS	Archivo con código C#.
.DLL	Librería de código.

Tabla 4. Archivos generados por cada **solución** o **proyecto** que vayamos a generar en Visual Studio

WINDOWS API

La interfaz de programación de aplicaciones de Windows es un conjunto de funciones residentes en bibliotecas (generalmente dinámicas, también llamadas DLL) que facilita el intercambio de mensajes o datos entre dos aplicaciones. De esta forma, permite que las aplicacioens trabajen al mismo tiempo –como podría ser un procesador de texto y una hoja de cálculo–, se comuniquen e intercambien datos.

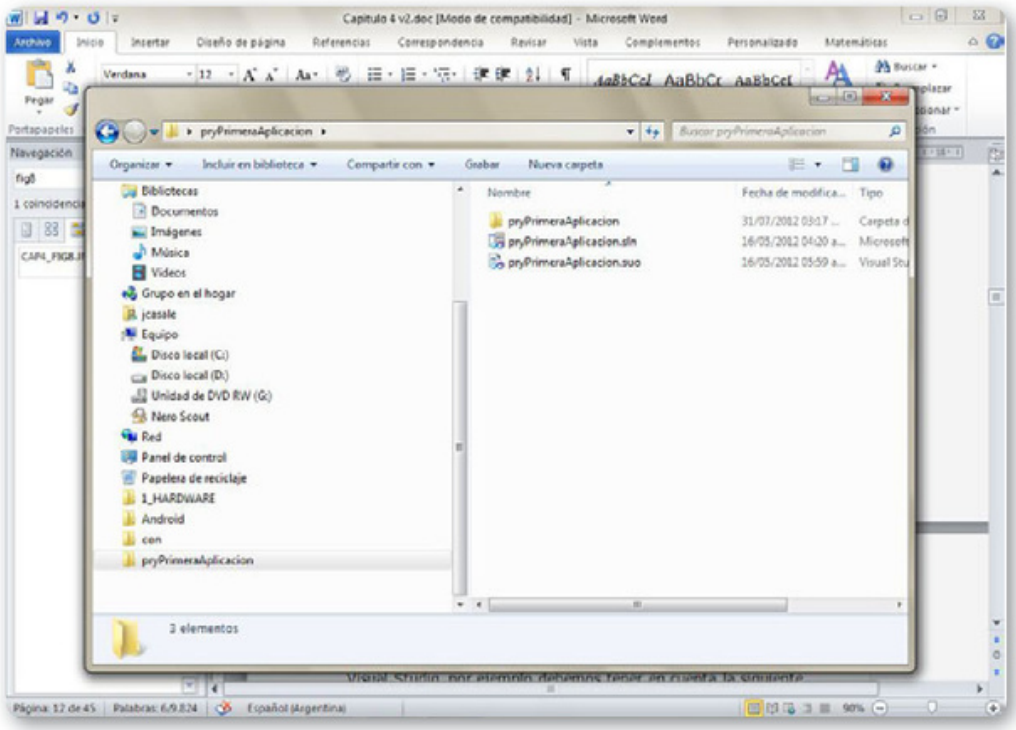


Figura 2. Al guardar, esta es la estructura de almacenamiento de archivos y carpetas de la solución o proyecto que se crea.

Podemos observar que cada proyecto es una única aplicación almacenada en su propia carpeta. Dentro de ella se encuentra el archivo de configuración del proyecto y los archivos reales XML, que contienen las referencias a todos los elementos, formularios y clases, además de las opciones de compilación.

Tenemos que considerar que en Visual Studio cada objeto o componente tiene **propiedades**, **métodos** y **eventos**. Por ejemplo, el contenedor por defecto en los proyectos **Windows Form** (que son para

MVA

Microsoft Virtual Academy es una plataforma de estudio creada por Microsoft que busca generar una experiencia de actualización y entrenamiento constante a todas aquellas personas interesadas en aprender sobre informática y tecnología. Su intención es maximizar el potencial de los interesados, simulando una academia virtual que permite seleccionar carreras y acceder a mucha información.

aplicaciones de escritorio) posee las siguientes propiedades destacadas:

- **Name**: nombre que asignaremos al objeto o componente, por ejemplo, **pryPrincipal**.
- **Text**: texto que aparecerá en dicho componente.

En todos los controles encontraremos la propiedad **NAME**, que es muy importante debido a que en ella asignamos el nombre del control, en donde se recomienda aplicar la nomenclatura para él. Por ejemplo, si agregamos un control **TextBox**, en su propiedad **name** podríamos asignarle **txtEjemplo**. A medida que vayamos desarrollando los temas del lenguaje de programación, iremos conociendo las propiedades principales de los controles que utilicemos.

Podemos encontrar muchísima información gratuita en el MSDN de Microsoft o en la página MVA. También es altamente recomendable el curso de desarrollador 5 estrellas que ofrecen las siguientes páginas.

MVA: **www.microsoftvirtualacademy.com**

MSDN: **http://msdn.microsoft.com**

Qué son y cómo se usan las variables

Las variables son uno de los elementos más importantes en el desarrollo de programas y reciben el nombre de identificadores. Pueden ser **constantes**, **variables**, **procedimientos** y **funciones**. Antes de continuar con su declaración, debemos conocer cuáles son los **tipos de datos** que podemos utilizar en aplicaciones de Visual Basic.

DATOS

TIPO DE DATO	DESCRIPCIÓN	INTERVALO DE VALORES
Byte	Números enteros	0 a 255 (sin signo)
Boolean	Valores lógicos	True o False
Integer	Números enteros	-2.147.483.648 a 2.147.483.647 (con signo)
Long	Números enteros	-9.223.372.036.854.775.808 a 9.223.372.036.854.775.807 (9,2...E+18 †) (con signo)

Single (punto flotante de precisión sencilla)	Números decimales	-3,4028235E+38 a -1,401298E-45 † para los valores negativos; 1,401298E-45 a 3,4028235E+38 † para los valores positivos
Double (punto flotante de precisión doble)	Números decimales	-1,79769313486231570E+308 a -4,94065645841246544E-324 † para los valores negativos; 4,94065645841246544E-324 a 1,79769313486231570E+308 † para los valores positivos
String (longitud variable)	Cadena de caracteres	0 a 2.000 millones de caracteres Unicode aprox.
Date	Fechas y horas	0:00:00 (medianoche) del 1 de enero de 0001 a 11:59:59 p.m. del 31 de diciembre de 9999.

Tabla 5. Tipos de datos en Visual Basic.

En el enlace de MSDN se muestran todos los tipos de datos que podemos encontrar: **http://msdn.microsoft.com/es-es/library/47zceaw7(v=vs.100).aspx**.

Ahora crearemos un nuevo proyecto en donde aplicaremos las distintas variables que estuvimos estudiando. En función del paso a paso anterior, en donde comenzamos con la creación del proyecto, aquí vamos a trabajar con algunas propiedades de la interfaz y el código fuente correspondiente a este lenguaje.

Para hacerlo, abrimos el proyecto **pryPrimeraAplicacion** y realizamos lo siguiente en la interfaz gráfica, asignando las propiedades del Form que aparecen a continuación:

a. Propiedad NAME = frmPrincipal

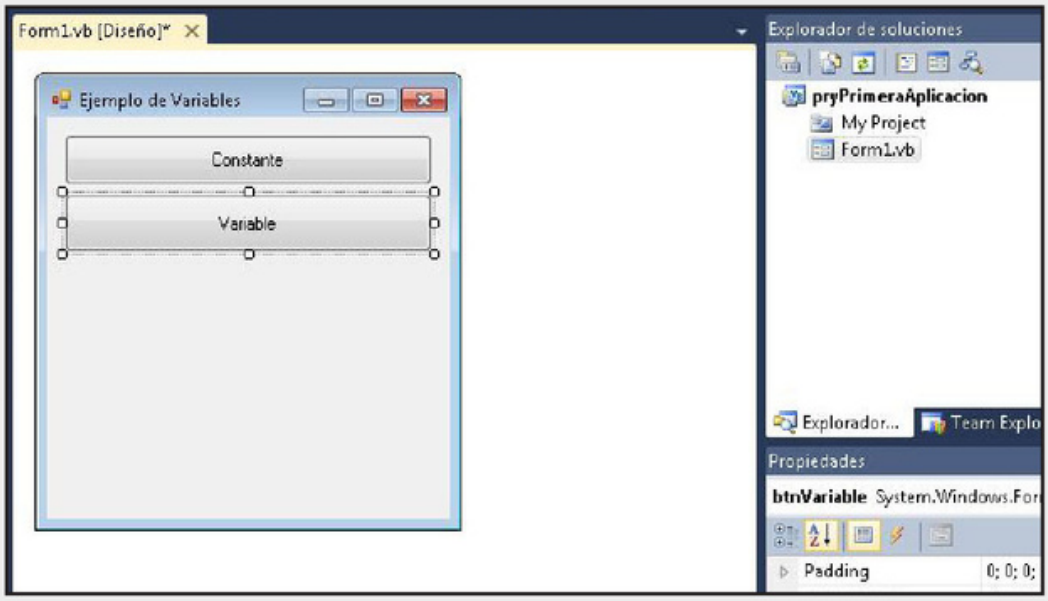
b. Propiedad Text = Ejemplo de Variables

TIEMPOS EN LA PROGRAMACIÓN

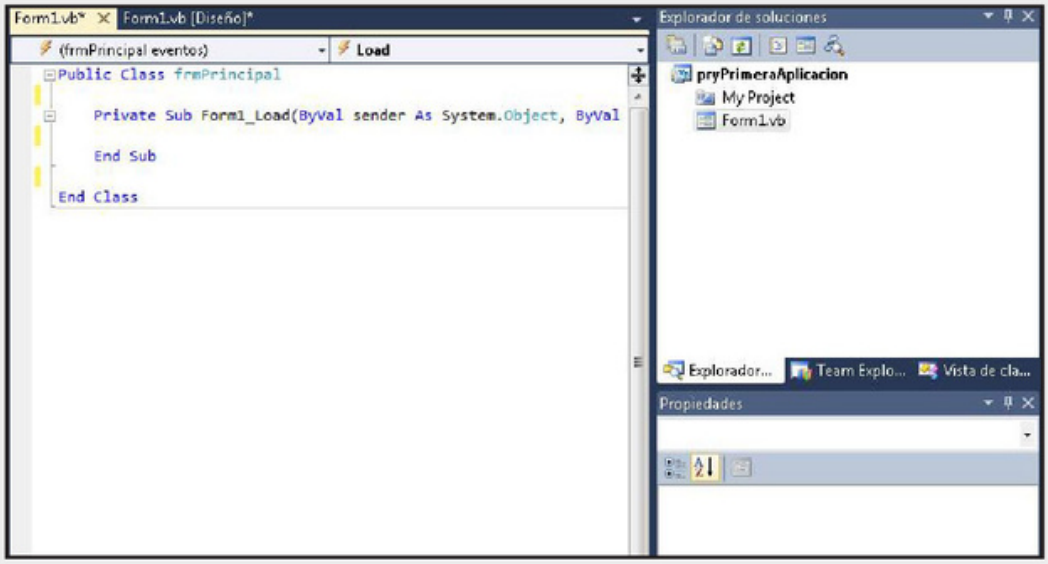
En la programación existen tres tiempos fundamentales. Primero, el tiempo de ejecución, momento en el que se ejecuta la aplicación y se llevan a cabo acciones sobre ella; segundo, el tiempo de diseño, en el cual el desarrollador asigna controles o formatos sobre la interfaz gráfica; y por último, el tiempo de código, cuando el desarrollador está trabajando en el código fuente.

PASO A PASO: PRIMEROS PASOS EN EL CÓDIGO

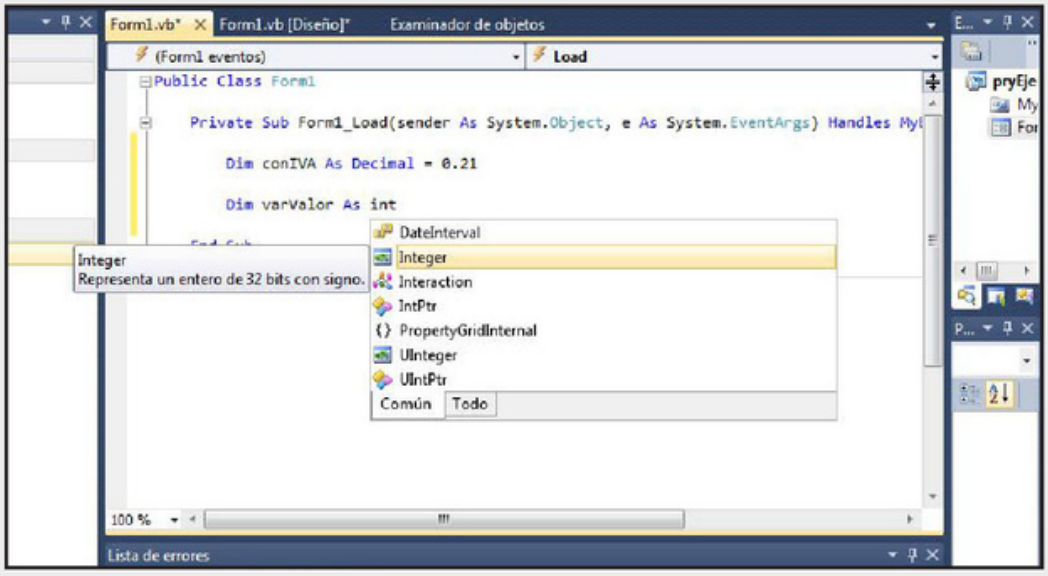
01 Agregue dos botones y asígneles: al control Button1, btnConstante (en Name) y Constante (en Text); y al control Button2, btnVariable (en Name) y Variable (en Text).



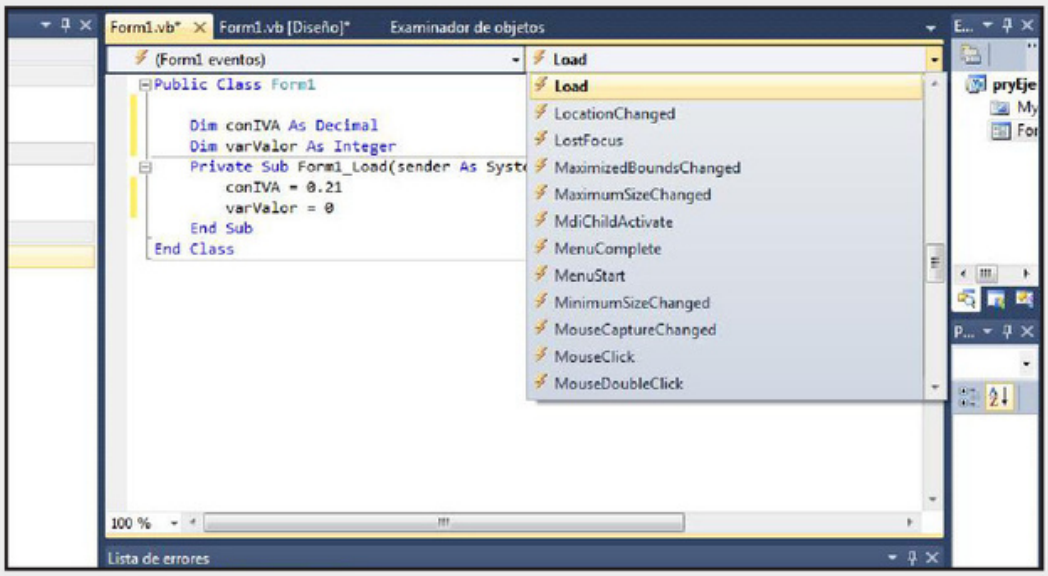
02 Seleccione el frmPrincipal y presione la tecla **F7**, o haga doble clic sobre él y entrará a tiempo de código (todo lo que hizo antes era en tiempo de diseño). De esta forma, verá que ha ingresado en un evento llamado Form1_Load.



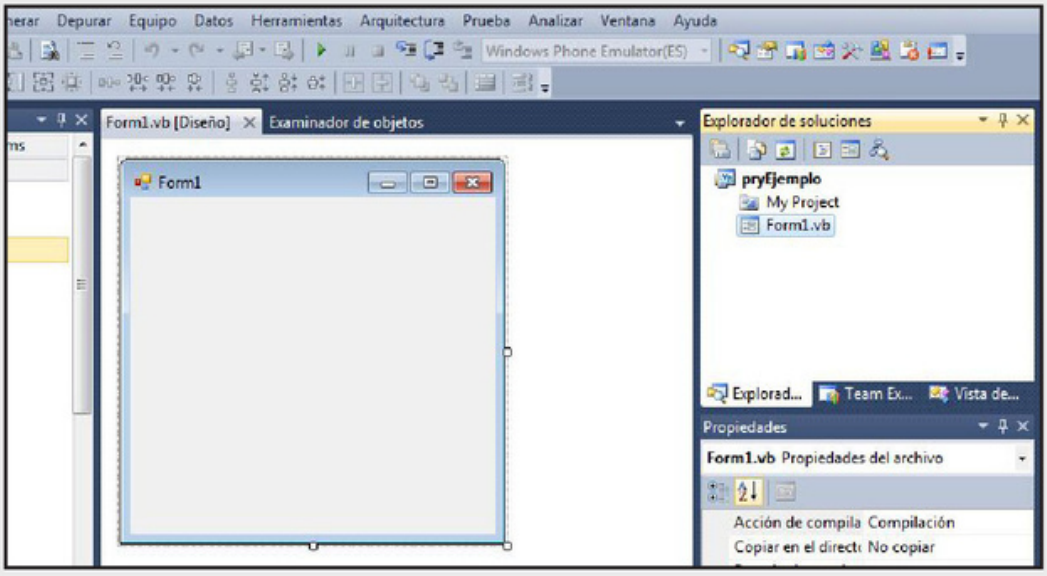
03 Dentro de este mismo código, cree las variables necesarias para el proyecto que se ven en la siguiente imagen.



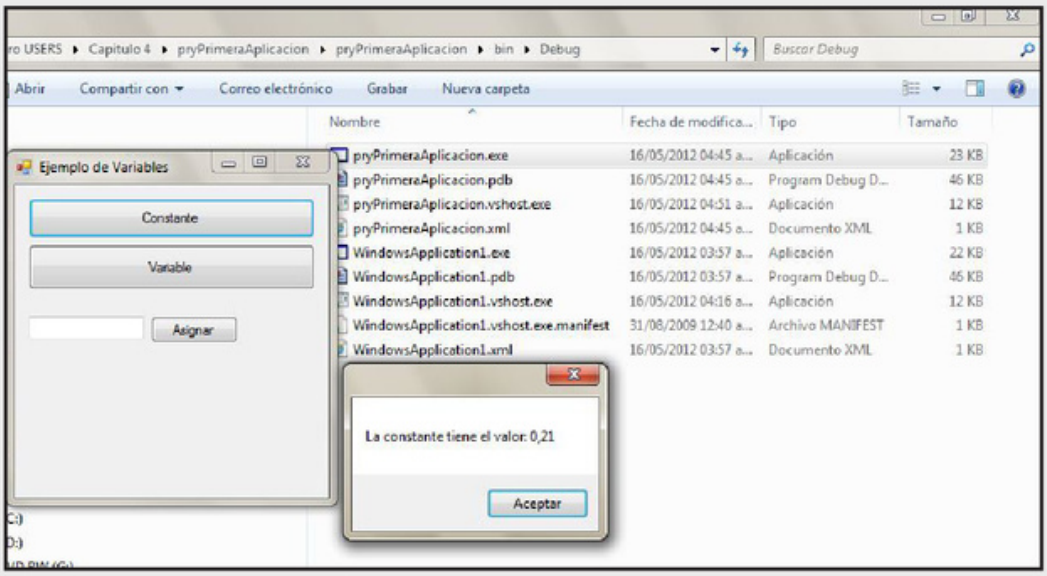
04 Vaya al modo de diseño, haga doble clic sobre el botón de comando de constante btnConstante, y acceda al tiempo de diseño de dicho botón. Por defecto, se mostrará el evento btnConstante_Click cuando el usuario lo ejecute. Para utilizar las variables en toda la aplicación, declárelas en el sector Declaraciones.



05 En el área para escribir código de btnConstante_Click, codifique un MessageBox para mostrar el valor de la constante. Al terminar de codificar, para probar la aplicación, presione F5 o vaya al menú Depurar/Iniciar Depuración, o haga clic en el botón Inicio.



06 Para concluir con el proceso, haga clic sobre el botón Constante y verá el siguiente resultado.



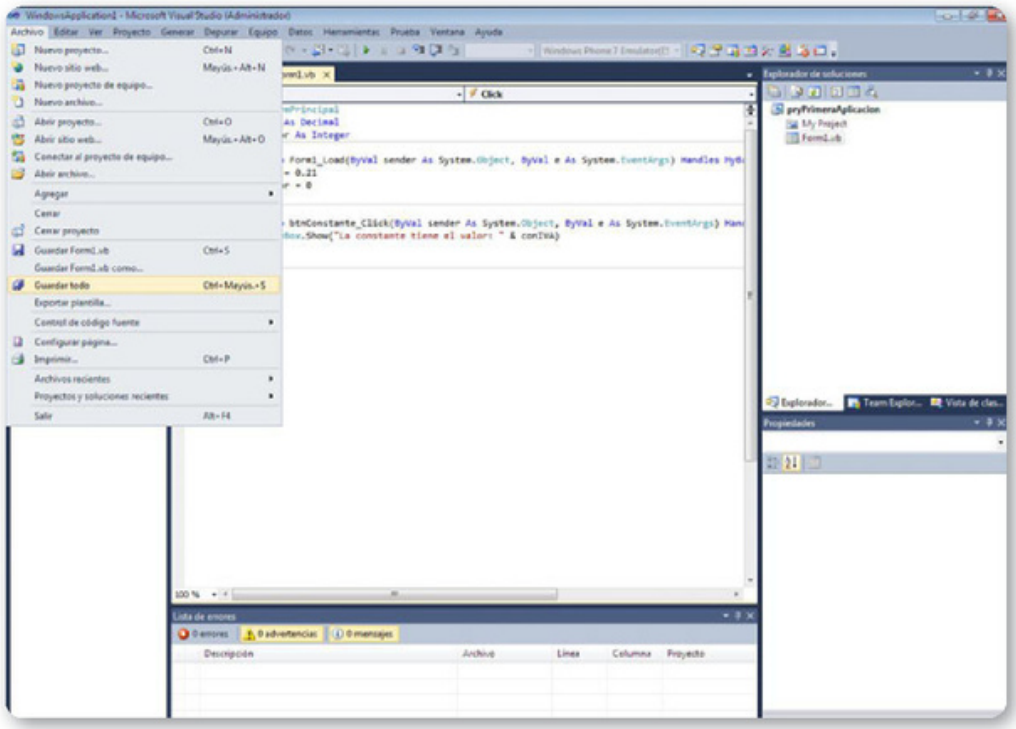


Figura 3. Seleccionando **Guardar todo** nos aseguramos de salvar todos los archivos que hayamos agregado al proyecto.

Al finalizar el ejemplo, desearemos guardar el proyecto. Sin embargo, para futuros proyectos es recomendable ir haciendo varias grabaciones parciales en forma constante, para asegurarnos de no perder ninguno de los cambios realizados.

Al momento de almacenar, podemos hacerlo desde el menú **Archivo/ Guardar todo**, seleccionando el icono desde la barra de herramientas o pulsando las teclas **CTRL+MAYUS+S**.

Ahora veamos cómo asignar valores a una variable, con el ejemplo **pryPrimeraAplicacion**. En el botón **btnVariable** hacemos doble clic con el botón izquierdo del mouse, y se nos enviará al código fuente en el evento **Click**. Allí escribimos lo siguiente:

```
varValor=150

MessageBox.Show("El valor de la variable es: " & varValor)
```

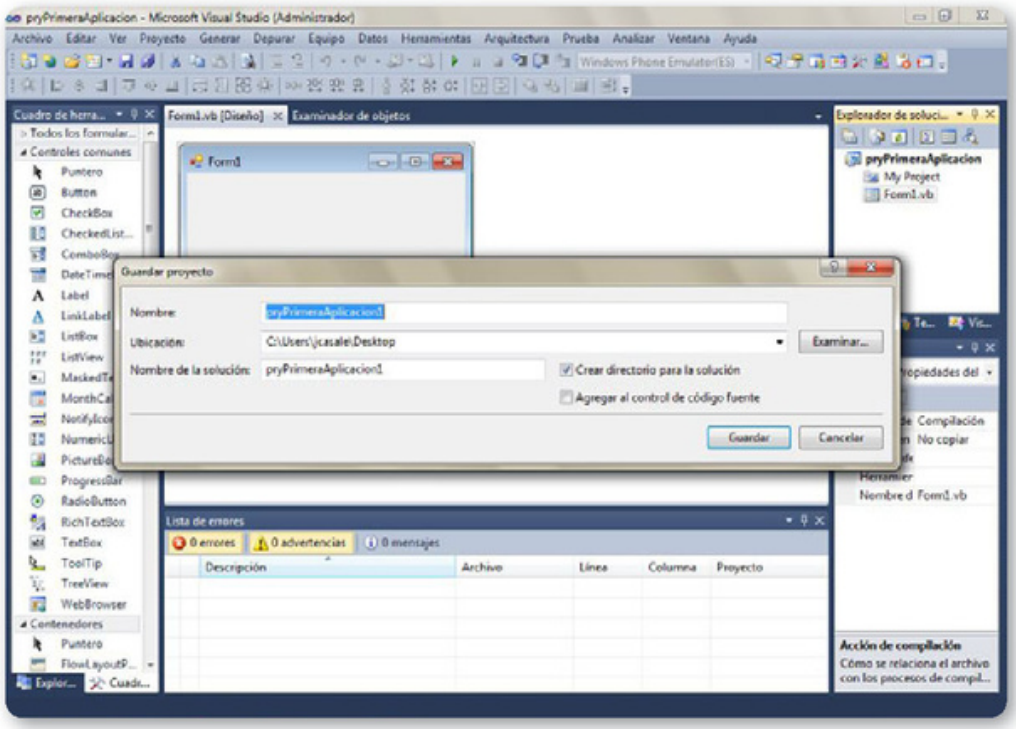


Figura 4. Al guardar, debemos controlar la ubicación y marcar **Crear directorio para la solución** para agrupar todo.

Para continuar con el ejemplo, asignaremos diferentes valores en tiempo de ejecución, es decir que el usuario pondrá un valor que cargaremos a **varValor**. Para esto, agregaremos en el proyecto:

1. Un control caja de texto (**TextBox**), al que le asignaremos la siguiente propiedad:
   a. **Name: txtCarga**
2. Al lado de **txtCarga**, un botón de comando (**Button**) con las siguientes propiedades:
   a. **Name: btnAsignar** b. **Text: Asignar**

PROPIEDAD TEXT

Una de las propiedades más frecuentes en los controles de Visual Studio es TEXT. Con ella podemos especificar o determinar el texto mostrado en un control, en especial, los de tipo Label y TextBox. Así también, en los Form de Windows Forms determina la barra de título correspondiente.

3. El código que aparece a continuación nos servirá para asignar valores a la variable:

```
Private Sub btnAsignar_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles btnAsignar.Click

    varValor = txtCarga.Text
    txtCarga.Text = ""
    MessageBox.Show("Se asignó el valor: " & varValor)

    End Sub
```

4. De esta forma, el funcionamiento será el siguiente: primero ingresar un dato numérico en **txtCarga** y, luego, hacer clic en **btnAsignar**, donde se muestra el valor que tiene **varValor** y blanquea cualquier dato que esté en **txtCarga.**

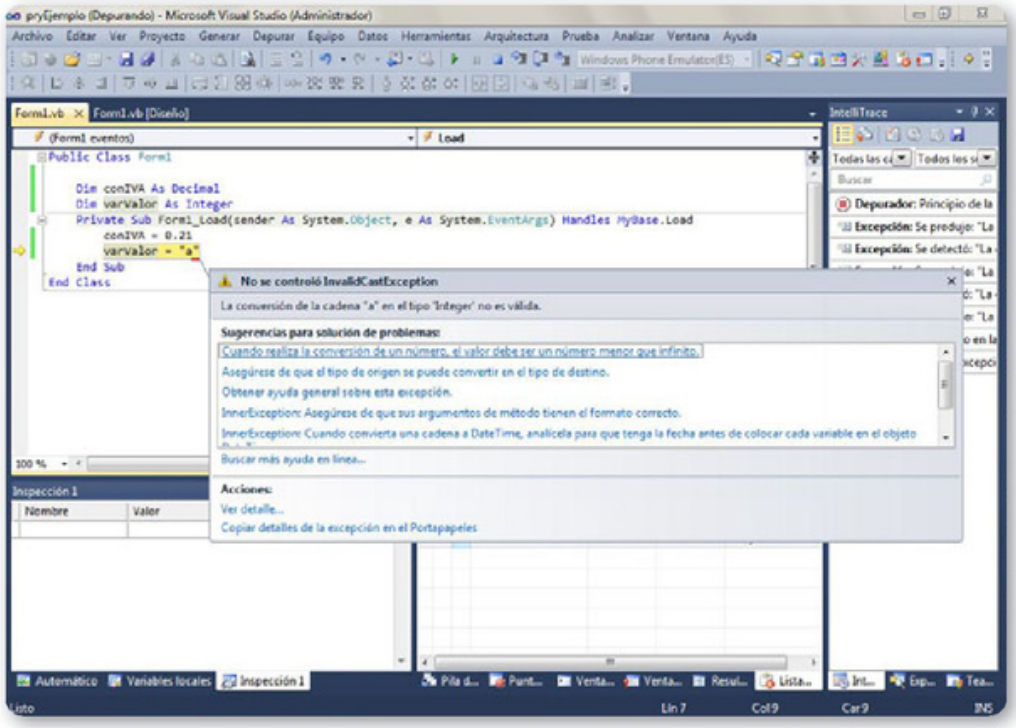


Figura 5. El error de conversión aparece cuando ingresamos un valor que no sea numérico o dejamos la caja de texto en blanco.

EL CARTEL DE ERROR SEÑALA QUE EL DATO NO CORRESPONDE A LA VARIABLE INDICADA

Este error indica que estamos intentado grabar un tipo de dato que no corresponde a la variable declarada, en este caso, un texto en una variable de tipo numérico.

Revisando el ejemplo que utilizamos, podemos determinar cómo es la sintaxis en Visual Basic para la declaración de variables:

```
[Public | Private] Const Nombre [As Tipo] = Expresión
[Public | Private] Dim Nombre [As Tipo]
```

- **Public** es opcional, e indica que la constante es pública y está disponible en todos los módulos.
- **Private** es opcional, e indica que la constante es privada y está disponible en el módulo donde se declaró.

Como podemos apreciar, Visual Basic tiene una palabra reservada para las constantes, que es **Const**. En este caso, debemos modificar la declaración de la constante de nuestro primer proyecto. Así, deberíamos escribir el siguiente código en la zona de declaraciones:

```
Const conIVA As Decimal = 0.21
```

Cómo se utilizan los operadores

Aritméticos

Como vimos en capítulos anteriores, los operadores aritméticos se utilizan para cálculos matemáticos, y van acompañados de variables y constantes para formar expresiones que retornan un valor. Por ejemplo, sumar el sueldo base con la remuneración y restarle los aportes.

MESSAGEBOX

La instrucción MessageBox como clase –es decir, que podemos utilizar en cualquier lenguaje que emplee el framework .NET– muestra un cuadro de mensaje que puede contener texto, botones y símbolos, para informar e instruir al usuario de una aplicación.

ARITMÉTICOS

OPERACIÓN	OPERANDO
Suma	+
Resta	-
Multiplicación	*
División	/
División entera	\
Módulo	Mod (resto de la división)
Exponenciación	^

Tabla 6. Estos son los operadores aritméticos que se utilizan en Visual Basic para realizar muchas de las operaciones aritméticas habituales que implican el cálculo de valores numéricos.

Utilizando el ejemplo, ahora veamos cómo debería ser el código en Visual Basic para realizar un cálculo determinado.

En este caso, nuestro objetivo será calcular un sueldo base, que es $4000 (cuyo valor puede variar); un aporte del 13% del sueldo base (este valor será fijo) y, finalmente, una remuneración por $500 (este valor puede variar).

```
Dim varSueldoBase As Integer = 4000
Const conAporte As Decimal = 0.13
Dim varRemuneracion As Integer = 500
Dim varSueldoNeto As Integer = 0

varSueldoNeto = varSueldoBase + varRemuneracion - (varSueldoBase
* conAporte)

MessageBox.Show("El sueldo neto es: " & varSueldoNeto)
```

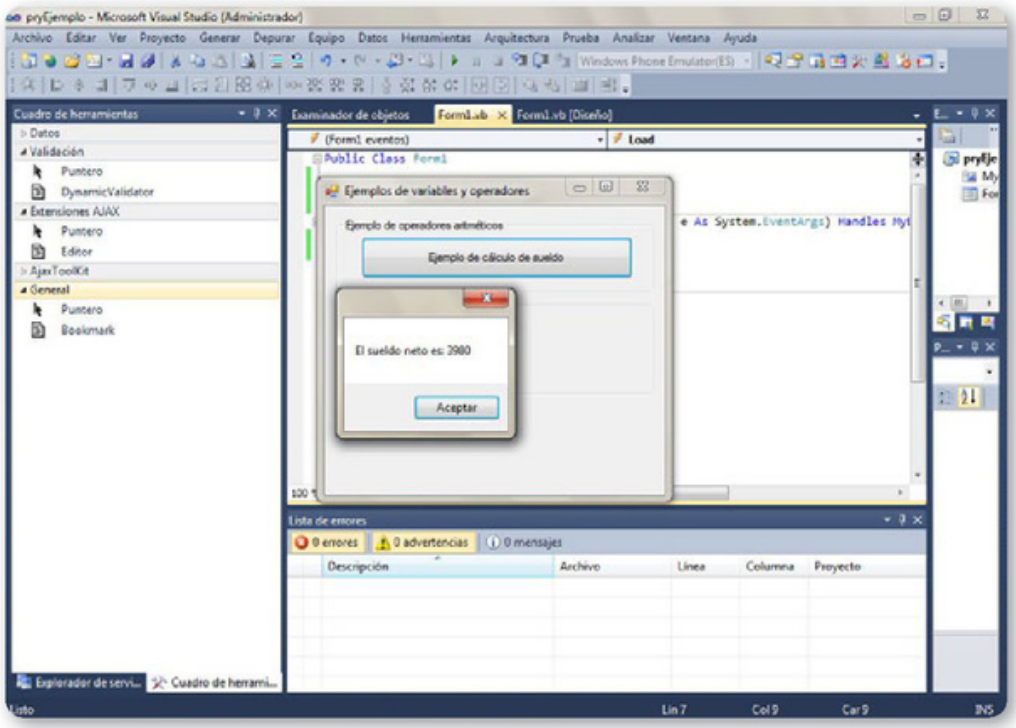


Figura 6. Interfaz y resultado del ejemplo, donde se muestra, en una ventana de diálogo, el código que generamos.

Lógicos

Al igual que los ejemplos de capítulos anteriores, los operadores lógicos permiten en el lenguaje conectar expresiones y determinar su veracidad o falsedad. Producen resultados de tipo **verdadero** o **falso**.

LÓGICOS

OPERACIÓN	DESCRIPCIÓN
And	Conjunción lógica
Or	Disyunción lógica
Not	Negación

Tabla 7. Operadores lógicos en Visual Basic.

En el ejemplo que veremos a continuación, vamos a comparar variables booleanas (que almacenan Verdadero o Falso) y utilizar las operaciones lógicas para ver los resultados que podemos obtener.

```
Dim varD1, varD2, varD3 As Boolean

varD1 = True
varD2 = True

varD3 = varD1 And varD2

MessageBox.Show("El resultado es: VERDADERO")
```

Recordemos que en el operador **AND** (como lo vimos en pseudocódigo) deben cumplirse todas las condiciones para que su resultado sea verdadero. En el próximo ejemplo veamos el uso de **OR**:

```
Dim varD1, varD2, varD3 As Boolean

varD1 = True
varD2 = False

varD3 = varD1 Or varD2

MessageBox.Show("El resultado es: VERDADERO")
```

Podemos revisar en el capítulo anterior, dentro de los operadores lógicos, la tabla donde comparamos las diferentes posibilidades y sus resultados. En el caso de Visual Basic, el uso es igual.

Relacionales

En el lenguaje, los operadores de comparación son utilizados para realizar comparaciones entre expresiones. A continuación, detallamos los operadores provistos por el lenguaje.

RELACIONALES	
▼ OPERACIÓN	▼ DESCRIPCIÓN
=	Igualdad
<>	Desigualdad
<	Menor que
>	Mayor que
<=	Menor o igual que
>=	Mayor o igual que
=	Igualdad

Tabla 8. Operadores relacionales.

Los operadores de comparación son utilizados en las estructuras de control de un programa y los veremos en el próximo tema a desarrollar; pero antes, es importante conocer los signos de concatenación. Si recordamos, antes definimos concatenar como la acción de unir dos expresiones alfanuméricas en un solo elemento. Por ejemplo, en Visual Basic utilizaremos:

```
Variable1 = "Juan" + " " + "Carlos"
Variable 2 = "XYZ" & 666
```

En el primer caso, a **Variable1** se le asignan las cadenas "Juan" espacio "Carlos". En el segundo caso, a la **Variable2** se le asigna la cadena "XYZ" y el valor entero 666.

Otro operador utilizado en el lenguaje es el operador **punto**, que sirve para conectar objetos con **propiedades** y **métodos**, recuperar y almacenar datos en variables creadas a partir de tipos de datos definidos por el usuario.

Ejemplos:

```
TxtDomicilio.Text = "BUENOS AIRES"
Persona.Edad = 30
```

En el lenguaje, el operador "=" se emplea para comparar y asignar. Ejemplos:

```
If a = b Then
    ...
End If

d = e + 47
```

Todo tiene un orden en la programación

Como vimos en pseudocódigo, para realizar un algoritmo debemos utilizar estructuras de control que nos permitan preguntar por una condición o realizar acciones repetidas veces. A continuación, veremos cómo debe ser su aplicación en el código fuente:

Estructuras de control

Las estructuras de operación de programas constituyen un grupo de formas de trabajo que, mediante el manejo de variables, nos permiten realizar ciertos procesos específicos para solucionar los problemas.

- **Estructura condicional**: las estructuras condicionales comparan una variable con otros valores para que, sobre la base del resultado

LIBRERÍAS (DLL)

Las DLL (*Dynamic-Link Library*, o biblioteca de enlace dinámico) son archivos que tienen en su estructura un código ejecutable cargado bajo la demanda de un programa informático por parte del sistema operativo. Por ejemplo, el caso de Microsoft Windows.

de esta comparación, se siga un curso de acción dentro del programa. En Visual Basic vamos a utilizar las siguientes palabras reservadas para estructuras condicionales: **If-Then-End If** y **Select Case**.

- **Simple**: la estructura **If-then-End If** permite ejecutar instrucciones en forma condicional, donde tendremos una condición que se puede cumplir o no, y el resultado será verdadero o falso. Dependiendo de esto, tendremos que realizar una o varias instrucciones de código. A continuación, compararemos la sintaxis en pseudocódigo con la de Visual Basic, para tomar como ejemplo.

Pseudocódigo	Visual Basic

Si <condición> entonces
Instrucción/es
Fin Si
	If condición then
	Instrucción/es
	End If

Instrucción: son las acciones por realizar cuando se cumple o no la condición.	If: indica el comando de comparación. Condición: es una expresión que se analiza y, en caso de ser verdadera, se ejecuta el bloque de instrucciones comprendido entre las palabras If y End if.

Ahora compararemos los ejemplos que teníamos en pseudocódigo, transformándolo a código de Visual Basic.

Ejemplo: Contamos con la variable Edad, en la cual el usuario ingresará un valor, y nosotros debemos preguntar si este es mayor o igual que 18. Si el resultado es verdadero, debe aparecer un mensaje que indique "Mayor de edad".

TECLAS RÁPIDAS

Para ir experimentando las teclas rápidas, revisemos un listado que nos será muy útil en programación:

F2: examinador de objetos

F5: ejecuta el proyecto

F8: ejecuta paso a paso (sentencia a sentencia)

F9: punto de depuración

CTRL+ENTER: detiene la ejecución de un programa

Para el siguiente ejemplo, asignaremos el valor 19 a la variable Edad:

Pseudocódigo	Visual Basic

variable Edad tipo numero
Edad ß 19
 	
Si Edad>=18 entonces
MOSTRAR "Mayor de edad"
Fin Si
	Dim Edad as Integer
	Edad=19
	
	If Edad>=18 then
	MessageBox.Show("Mayor de edad")
	End If

Como podemos observar, las estructuras en sí son muy parecidas; solo debemos estar atentos a la nomenclatura del lenguaje que utilicemos para nombrar sus condicionales simples. Cabe destacar que cuando escribimos **IF** en Visual Basic y presionamos dos veces la tecla **TAB**, automáticamente se completa todo el bloque **IF-THEN-END IF** sin necesidad de escribirlo. De esta forma, aparecerá el siguiente texto:

```
If True Then

End If
```

Vemos que escribe automáticamente **True**, en donde debemos ingresar la condición que deseamos corroborar.

- **Doble**: las estructuras condicionales dobles permiten elegir entre dos opciones, en función del cumplimiento o no de una determinada condición. En Visual Basic tienen la siguiente sintaxis:

```
If condición Then
    Instrucción/es
Else
    Instrucción/es
End If
```

De esta forma, podemos realizar acciones si las opciones de resultado son verdaderas o falsas. Continuando con el ejemplo anterior, indicaremos cuándo la variable Edad es mayor que 18 y cuándo no.

Pseudocódigo	Visual Basic

variable Edad tipo numero
Edad ß 19
 	
Si Edad>=18 entonces
MOSTRAR "Mayor de edad"
Sino
MOSTRAR "Menor de edad"
Fin Si
	Dim Edad as Integer
	Edad=19
	
	If Edad>=18 then
	MessageBox.Show("Mayor de edad")
	Else
	MessageBox.Show("Menor de edad")
	End If

- **Múltiples o anidadas**: las estructuras de comparación múltiples o anidadas son decisiones especializadas que nos permiten comparar una variable y sus posibles valores. Dependiendo de estos valores, se ejecutará el bloque de instrucciones apropiado. La sintaxis de estas estructuras es la siguiente:

```
If condición then
Instrucción/es
Else
If condición then
Instrucción/es
Else
                If condición then
Instrucción/es
Else
Instrucción/es
End If
End If
End If
```

Por ejemplo: deseamos saber si la variable **varIntervalo** está comprendida entre ciertos valores. Dependiendo de esto, indicaremos que si **varIntervalo** está entre 0 a 20, es un valor "bajo"; entre 21 y 50 es "medio"; entre 51 y 80 es "alto"; y entre 81 a 100 es "excesivo". A continuación, veamos la codificación correspondiente, suponiendo que el usuario ingresa el valor 66:

```
varIntervalo = 66

If varIntervalo >=0 AND varIntervalo <=20 then
        MessageBox.Show("Valor Bajo")
Else
    If varIntervalo >=21 AND varIntervalo <=50 then
        MessageBox.Show("Valor Medio")
    Else
        If varIntervalo >=51 AND varIntervalo <=80 then
            MessageBox.Show("Valor Alto")
        Else
            MessageBox.Show("Valor Excesivo")
        End If
    End If
End If
```

Otras de las estructuras que podemos utilizar para realizar este tipo de condicionales múltiples es el **select case**, que en pseudocódigo vimos como **según sea**. Veamos la sintaxis en ambos:

Pseudocódigo	Visual Basic
Segun_Sea <condición> hacer Caso 1: Instrucción/es Caso 2: Instrucción/es Caso 3: Instrucción/es Sino Instrucción/es Fin_Segun	Select Case Variable Case Constante1 Instrucción1() Case Constante2 Instrucción2() Case Else InstrucciónN() End Select

Apliquemos esta instrucción para resolver el ejemplo anterior:

```
Dim varNumero As Integer

varNumero = 66

Select Case varNumero
        Case Is <= 21
        MessageBox.Show("El valor es Bajo")
        Case Is <= 51
        MessageBox.Show("El valor es Medio")
        Case Is <= 81
        MessageBox.Show("El valor es Alto")
        Case Else
        MessageBox.Show("El valor es Excesivo")
End Select
```

Podemos decir que la estructura condicional múltiple **Select Case** es un **If** anidado más desenvuelto, desde el punto de vista de la claridad del código que escribimos. Para este ejemplo, también podemos utilizar la siguiente forma de comparación:

```
Select Case varNumero
        Case Is < 21
        MessageBox.Show("El valor es Bajo")
        Case 21 to 50
        MessageBox.Show("El valor es Medio")
        Case 51 to 80
        MessageBox.Show("El valor es Alto")
        Case Else
        MessageBox.Show("El valor es Excesivo")
End Select
```

Hasta aquí revisamos las estructuras de control utilizadas para preguntar secuencialmente sobre alguna condición o caso. En este lenguaje de programación utilizaremos **If-Else-End If** para condiciones

simples o múltiples, y la estructura **Select Case**, que es muy útil para algunos tipos de condiciones múltiples.

- **Repetitivas o estructuras cíclicas**: desde el lenguaje de programación, vamos a repasar cómo podemos repetir partes de un programa mientras cierta condición se cumpla o sea verdadera, y conoceremos las distintas estructuras que podemos utilizar.
  La estructura de control repetitiva **Do Loop** le permite a nuestro desarrollo reiterar la ejecución de un bloque de instrucciones hasta que se cumpla cierta condición. La sintaxis tiene dos variantes:

```
Do While condición
        Instrucciones()
Loop
```

En este caso se analiza la condición. Si es verdadera, se ejecutará el bloque de instrucciones delimitado entre **Do** y **Loop**, y el proceso se repetirá otra vez, siempre que el resultado de la condición sea verdadero.

```
Do
        Instrucciones ()
Loop While Condición
```

Como podemos ver, esta última estructura es muy similar a la anterior, solo que cambia la ejecución de las instrucciones. En el bucle anterior primero PREGUNTA sobre la condición dada y luego HACE; en cambio, en esta estructura, primero HACE y luego PREGUNTA. En pseudocódigo equivaldría a la sintaxis **Mientras que – Fin mientras**.

La diferencia que existe entre estas estructuras es el orden en el que se ejecutan los bloques de instrucciones algorítmicas. En un caso se analiza la condición y luego se ejecuta el bloque de instrucciones, y en el otro se ejecuta el bloque de instrucciones y, luego, se analiza la condición. Otras estructuras repetitivas que podemos encontrar son:

> LA DIFERENCIA DEPENDERÁ DEL ORDEN QUE EXISTA EN LOS BLOQUES DE INSTRUCCIÓN

```
Do Until Condición
        Instrucciones()
Loop
```

En esta estructura se analiza la condición. Si es falsa, se ejecuta el bloque de instrucciones y el proceso se repite hasta que la condición se vuelva verdadera.

```
Do
        Instrucciones()
Loop Until Condición
```

Para esta instrucción repetitiva, primero se ejecuta el bloque de instrucciones y, luego, se analiza la condición. Si se trata de una condición falsa, el proceso se repetirá otra vez hasta lograr que la condición sea verdadera.

Veamos ahora una estructura repetitiva que nos resultará muy útil, que en pseudocódigo equivaldría a la sintaxis **Hacer-Hasta**. En el caso de este lenguaje, se utiliza **For** o **For Each**.

La instrucción **For-Next** ejecutará en un determinado número de veces un bloque de código. Veamos la sintaxis:

```
For Variable = Valor1 To Valor2 Step Incremento
        Instrucciones()
Next
```

El bloque de instrucciones que se repite está delimitado por las instrucciones **For** y **Next**. La sintaxis **Variable** es una variable de tipo numérico que toma valores entre **Valor1** y **Valor2**. Por cada vez que el bloque se ejecuta, el valor de **Variable** se incrementa en el valor especificado en **Incremento**, que puede ser positivo o negativo.

A continuación, veamos un ejemplo que nos demuestre cómo sumar de un número 5 veces:

```
Dim i, respuesta As Integer

For i = 1 To 5
    respuesta = i + i
Next
```

En este ejemplo, la repetitiva ejecutará 5 veces desde i=1 hasta 5 y acumulará la suma en la variable **respuesta**. Si la variable i iniciara en 0, entonces se ejecutaría 6 veces.

Revisando estas estructuras, podemos ver que las repetitivas se aplican para realizar una acción, siendo necesario o no cumplir una condición determinada. A continuación, veamos las diferencias.

REPETITIVAS

ESTRUCTURA	DESCRIPCIÓN DE USO
Do While condición **Instrucciones()** **Loop**	Ejecutará tantas veces el bloque de código en función de una condición, ya sea verdadero o falso.
Do **Instrucciones ()** **Loop While Condición**	Primero ejecutará el bloque de instrucciones y luego evaluará la condición.
Do Until Condición **Instrucciones()** **Loop**	Ejecutará tantas veces el bloque de código en función de una condición, hasta que sea verdadera.
Do **Instrucciones()** **Loop Until Condición**	Primero ejecutará el bloque de instrucciones y luego evaluará la condición.
For Variable = Valor1 To Valor2 Step Incremento **Instrucciones()** **Next**	Realizará una acción, a veces en la condición, hasta cumplirla; las instrucciones serán ejecutadas cada vez que ingrese.

Tabla 9. Comparación y descripción de las estructuras repetitivas.

Como vimos hasta aquí, podemos bosquejar un algoritmo en pseudocódigo y, luego, volcarlo en la sintaxis del lenguaje de programación. Ahora revisaremos el uso de vectores y matrices.

Tipos de datos estructurados

Los tipos de datos estructurados son espacios en memoria que serán utilizados mientras la aplicación sea ejecutada y se borrarán de memoria al momento de finalizar el programa. Los vectores y matrices son las estructuras que utilizaremos para almacenar información de manera temporal y manipular estos datos.

Matriz

Las estructuras que se consideran una matriz son aquellas que tienen un conjunto de elementos relacionados lógicamente entre sí. Sus elementos deben ser referidos mediante un solo nombre y un número llamado **índice**, para así poder distinguirlos. Los elementos son seguidos desde el índice cero hasta el índice de valor superior. En la **Figura 7** vemos un ejemplo.

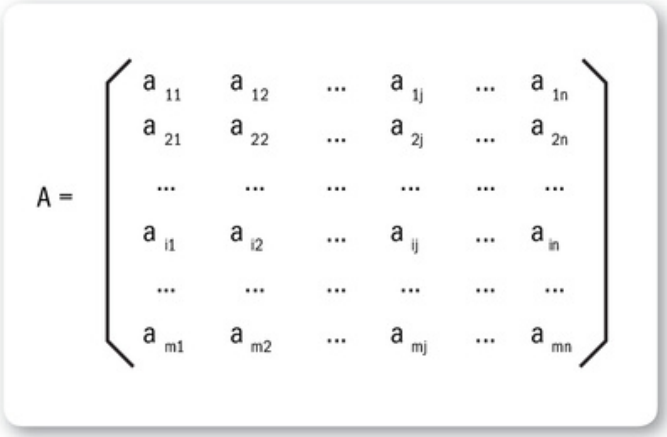


Figura 7. En esta matriz teórica podemos ver el elemento A compuesto por innumerables valores.

Cada uno de los números que integran la matriz se denomina **elemento**, y se distingue de los demás por la posición que ocupa; es decir, la **fila** y la **columna** a la que pertenece. Este tema lo desarrollamos en el capítulo dedicado al pseudocódigo; ahora veremos algunos ejemplos en el lenguaje de programación.

Declaramos una matriz para almacenar la cantidad de horas trabajadas por día de lunes a viernes:

```
Dim Dias(5) As Integer
```

Esta matriz tiene seis elementos, y los índices están en el rango de cero a cinco. Su declaración es más simple y útil que declarar seis variables diferentes, es decir, seis espacios en memoria donde se pueda asignar un valor numérico. La sintaxis para declarar una matriz es la siguiente:

```
Dim Vector(5) As Integer
```

Esta sería la sintaxis para declarar lo que conocemos como un vector, es decir, una matriz de una sola dimensión o unidimensional; es lo que usualmente llamamos **Vector**.

```
Dim Matriz(5, 5) As Integer
```

Esta sería la sintaxis para declarar una matriz de dos dimensiones o bidimensional:

```
Dim Matriz(5, 5, 3) As Integer
```

Esta sería la sintaxis para declarar una matriz de tres dimensiones, que son más complejas para manejar en código; se llaman **tridimensionales**.

QBASIC, UNO DE LOS MÁS USADOS

El QBASIC, originado por contracción del nombre del producto QuickBasic que se traduce a BASIC, es una variante del lenguaje de programación BASIC. Provee de un IDE avanzado, incluyendo un depurador con características tales como evaluación de expresiones y modificación de código.

Para asignar valores a una matriz, tan solo debemos indicar el espacio que deseamos utilizar, por ejemplo:

```
Dias(0) = 6
```

Esto quiere decir que en la posición 0 de Dias se almacenará el valor 6.

Cuando estamos manejando estas estructuras, también podemos asignar valores al declararlas, por ejemplo:

```
Dim Personas() As String = {"Marcio", "Cesar", "John", "Agustina"}
```

La cantidad de elementos asignados determina el tamaño de la estructura, que, según esta declaración, sería una matriz de una dimensión o un vector, que cuenta con 4 posiciones.

Nombre de Matriz: Personas				
Posición	0	1	2	3
Dato	Marcio	Cesar	John	Agustina

Figura 8. Declaración de una matriz simple; contiene 4 datos que son texto, y la posición inicia en 0.

- **Dimensión**: dentro de una matriz hacemos referencia a un índice que nos indica la dirección específica de elementos dentro de ella. Supongamos que tenemos una matriz que contiene el total de horas trabajadas por día. En este caso, crearemos una estructura que tenga como dimensión la cantidad de días, donde se guarden las horas trabajadas. Por otro lado, si tenemos la necesidad de guardar el total de horas trabajadas por empleado, debemos utilizar una matriz que tenga dos dimensiones de almacenamiento: los nombres de las personas y su total de horas trabajadas. A continuación, veamos los ejemplos pasados a código.

Ejemplo 1:

```
Dim HorasTrabajadas(n) As Integer
```

Ejemplo 2:

```
Dim TotalHorasTrabajadas(n, n) As Integer
```

N hace referencia a una cantidad que se especificará en la programación.

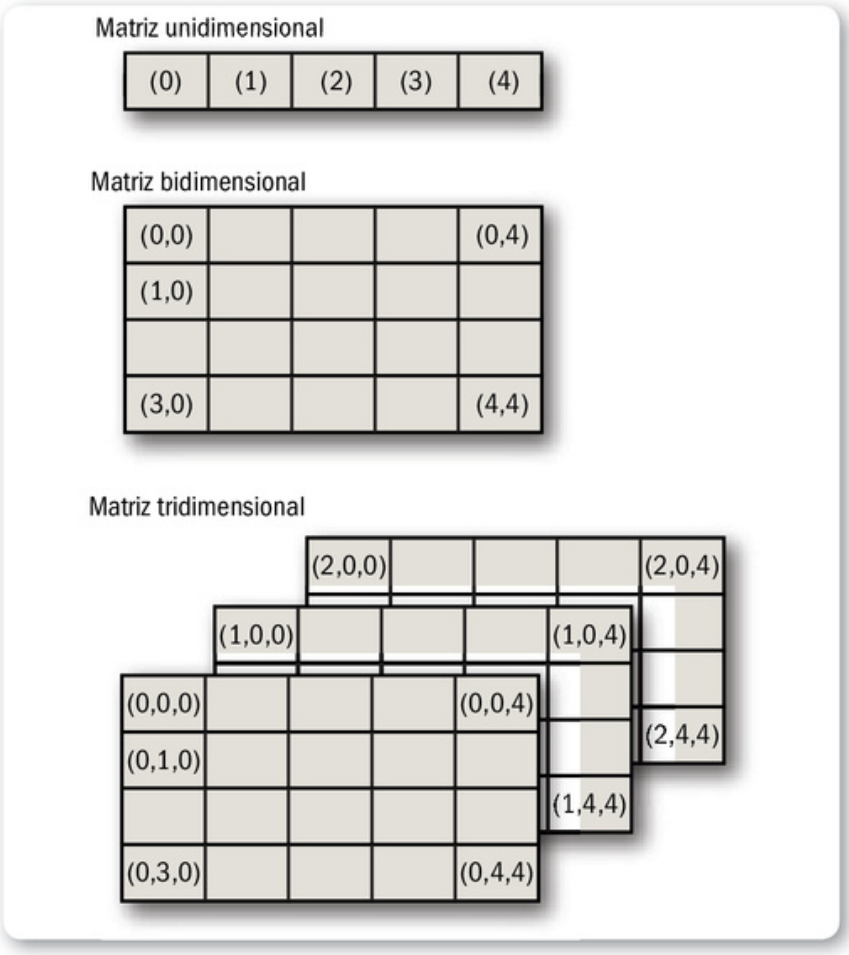


Figura 9. Dimensiones de una matriz. Diferentes representaciones de las matrices en la memoria de un equipo.

Si queremos utilizar más de tres dimensiones, esta solo podrá tener un máximo de **32 dimensiones**. Es importante que estemos atentos al agregar estas dimensiones, ya que el espacio total necesario irá aumentando de manera considerable.

Almacenamiento de información

Podemos crear matrices que no contengan información, o cargar información sobre ellas. Veamos cómo se crea una matriz sin elementos, declarando una de sus dimensiones en -1:

```
Dim matDosDimensiones (-1, 3) As Integer
```

A continuación, analizaremos algunas circunstancias en las cuales nos sería útil crear matrices de longitud cero:

- Deseamos que el código utilizado sea más sencillo, sin tener que comprobar **Nothing** como caso especial.
- El código interactúa con una interfaz de programación de aplicaciones (API) que exige pasar de una matriz de longitud cero a uno o más procedimientos, o devuelve una matriz de longitud cero desde uno o más procedimientos.

Conociendo las matrices de longitud cero, veamos cómo podemos rellenar una matriz con valores iniciales que antes utilizamos, pero aplicando ahora un literal de matriz. Este se encuentra formado por una lista de valores separados por comas que se encierran entre llaves **({})**.

Cuando se crea una matriz utilizando un literal de matriz, se puede proporcionar el tipo de la matriz o usar la inferencia de tipos para determinarlo. Ambas opciones se muestran en el código siguiente:

```
Dim matNumeros = New Integer() {1, 2, 4, 8}
Dim matValores = {"a", "valor", "b", "texto"}
```

Cabe destacar que el tipo de dato que se infiere por defecto es el **Object**; por lo tanto, si se declara una variable sin referenciar su tipo de dato, este tomará el tipo **Object**.

Las declaraciones explícitas del tipo de los elementos se crean utilizando un literal de matriz y los valores se deben ampliar. En el siguiente ejemplo de código podemos ver cómo se crea una matriz de tipo Double a partir de una lista de enteros:

```
Dim matDatos As Double() = {1, 2, 3, 4, 5, 6}
```

Hasta aquí hemos visto la asignación de valores en la declaración de estructuras; ahora veremos cómo recorrer una matriz y guardar información en ella. Vamos a utilizar el ejemplo del capítulo anterior, aplicado en pseudocódigo para el manejo de vectores:

Pseudocódigo	Visual Basic
Variable vecEjemplo (30) tipo numero Variable i tipo numero ß 0 Para i ß 1 hasta 30 hacer vecEjemplo(i) ß i Fin Para MOSTRAR "Vector cargado"	Dim vecEjemplo(30) As Integer Dim indice As Integer = 0 For indice = 1 To 30 vecEjemplo(indice) = indice Next MessageBox.Show("Vector Cargado")

En el ejemplo de Visual Basic creamos un vector ejemplo con 31 posiciones y cargamos los valores de la variable índice, con lo cual quedan cargados valores del 1 al 30. Para asignar valores en un sector específico de este ejemplo, podríamos realizar lo siguiente:

```
vecEjemplo(10) = 150
```

En el código anterior indicamos que, en la posición 10 de **vecEjemplo**, vamos a grabar 150. Veamos cómo mostrar esta posición u otra:

```
MessageBox.Show("El valor en la posición es: " & vecEjemplo(10))
```

Recorrido de información

Para ver la información secuencial de una estructura, ya sea unidimensional (vector) o de varias dimensiones, debemos realizar la siguiente codificación, que aquí compararemos con el pseudocódigo visto en el capítulo anterior:

Pseudocódigo	Visual Basic
Variable matEjemplo (4, 4) tipo texto	Dim matEjemplo (4, 4) As String
matEjemplo(1,1) ß "Lunes" matEjemplo(1,3) ß "Fuente" matEjemplo(3,2) ß 5000 matEjemplo(4,4) ß "Ultimo"	matEjemplo(1,1) = "Lunes" matEjemplo(1,3) = "Fuente" matEjemplo(3,2) = 5000 matEjemplo(4,4) = "Ultimo"
Variable ifila, icolumna tipo numero ß 0 Para ifila ß 1 hasta 4 hacer Para icolumna ß 1 hasta 4 hacer MOSTRAR matEjemplo(ifila, icolumna) Fin Para Fin Para	Dim ifila, icolumna As Integer For ifila = 1 to 4 For icolumna = 1 to 4 MessageBox.Show(matEjemplo(ifila, icolumna)) Next Next

Cabe destacar que estaremos mostrando mensajes por la cantidad de veces que entremos dentro de los datos de la matriz. En este caso, encontraremos que son 4 filas y 4 columnas, que nos darán un resultado de 16 espacios, constituyendo así 16 ventanas. Para evitar esto, podemos aplicar el código de ejemplo que aparece a continuación y, así, mostrar solo los espacios que contienen valor:

```
For ifila = 1 To 4

  For icolumna = 1 To 4
If matEjemplo(ifila, icolumna) <> Nothing Then
    MessageBox.Show(matEjemplo(ifila, icolumna))
```

```
        End If

    Next
Next
```

En el código anterior vimos solo los espacios que contenían valores distintos a nada (Nothing). De esta forma, tanto en pseudocódigo como en código del lenguaje, el recorrido de la información almacenada en una estructura debe iniciarse con un índice. Si la dimensión de dicha matriz fuese 3, podríamos crear índices X, Y y Z para recorrerlos.

Uso de controles básicos

La referencia que hacemos a controles básicos no es porque sean los más sencillos, sino por ser los más utilizados. En la **Figura 10** vemos como ejemplo los más frecuentes.

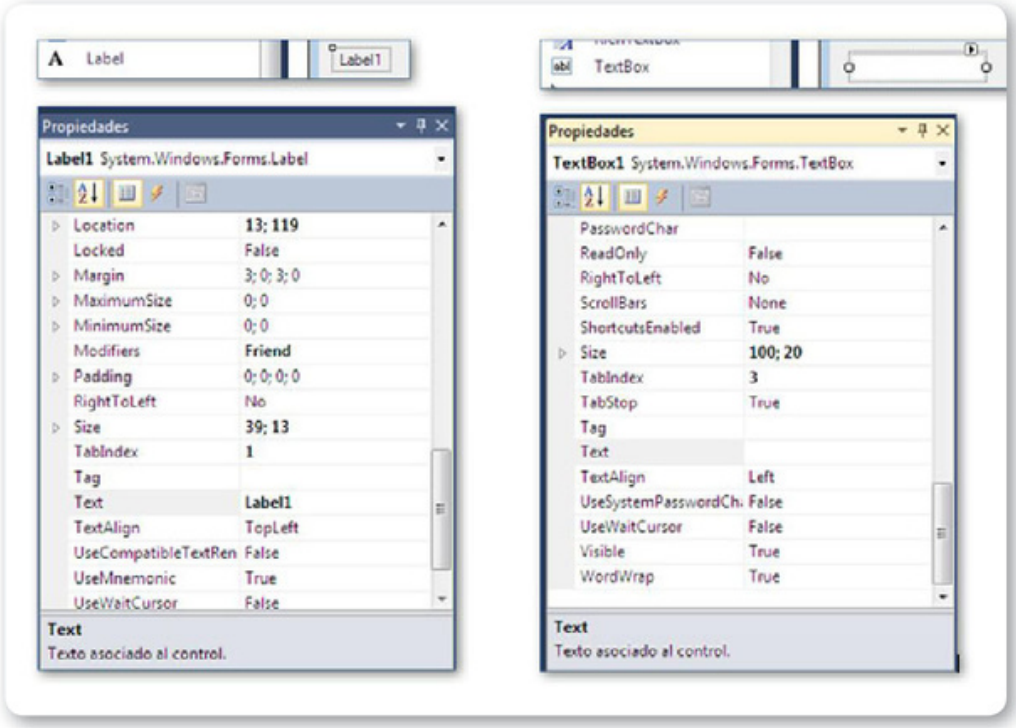


Figura 10. Controles comunes Label/TextBox, los más utilizados para mostrar e ingresar información en una aplicación.

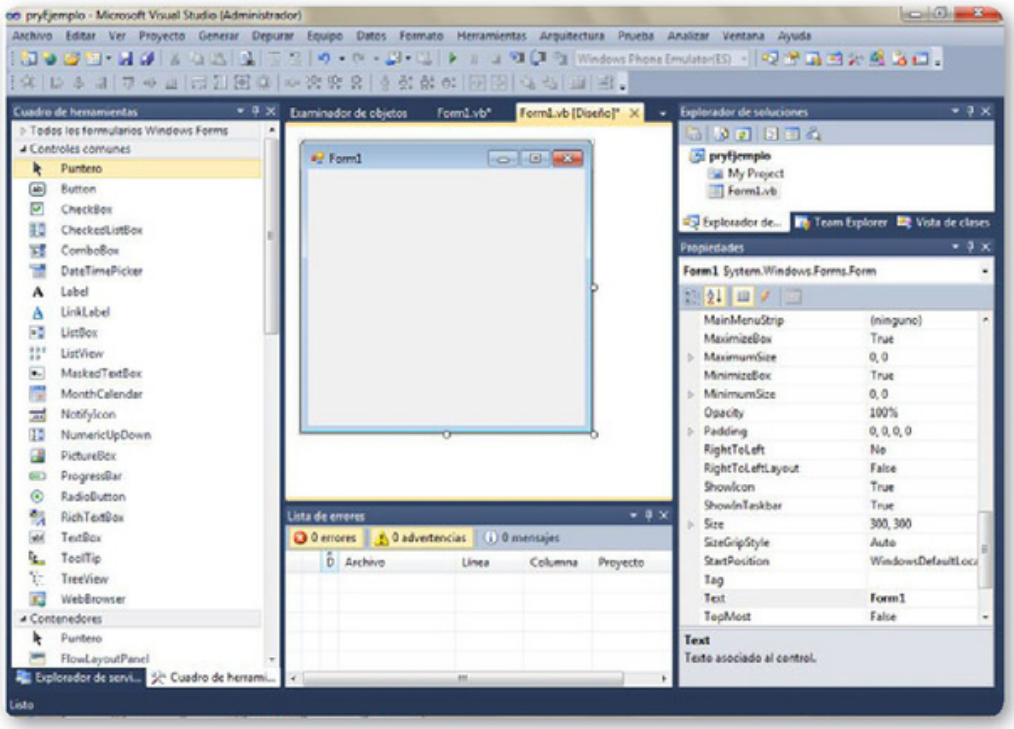


Figura 11. Forma por defecto que es representada en Visual Basic y tomada de Windows.Forms.

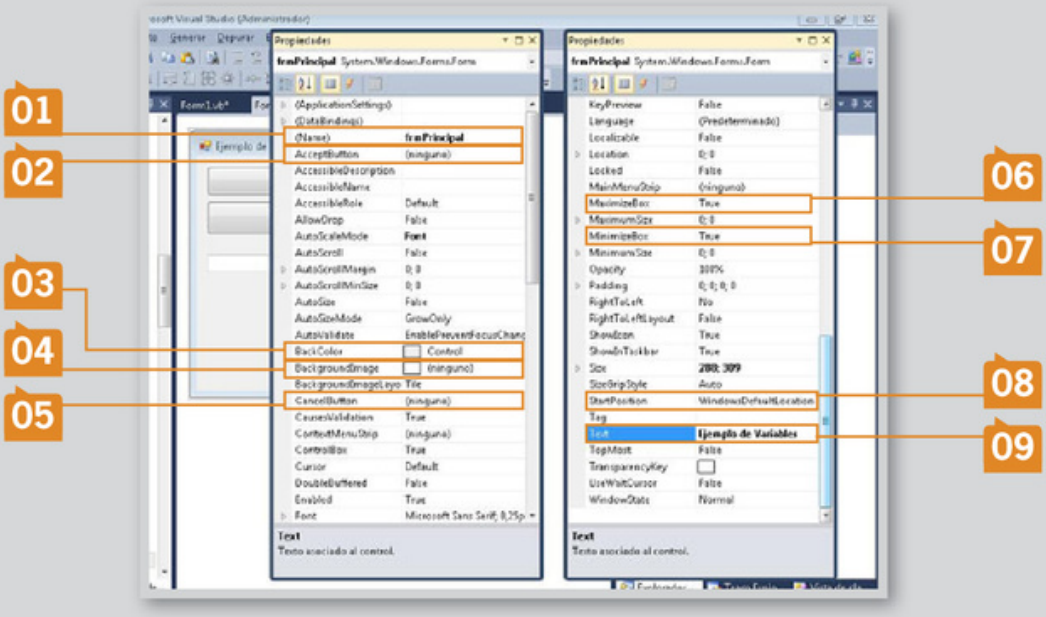
Gracias a estos controles, los usuarios pueden operar y obtener los resultados necesarios de una aplicación. De esta manera, se pueden añadir controles a un formulario, seleccionando la herramienta adecuada del cuadro correspondiente.

En función a los pasos que hemos vistos anteriormente, ahora es el momento de crear un nuevo proyecto. Este nos permitirá ver en detalle el funcionamiento y algunas de las características principales correspondientes al form (forma o formulario).

INFERENCIA DE TIPOS

Dicha inferencia es un proceso por el cual el compilador puede determinar el tipo de dato en una variable local, la que ha sido declarada sin explicitar su tipo. De esta forma, el tipo de dato es inferido a partir del valor inicial provisto a la variable. Por ejemplo, en Visual Basic se infiere el tipo Object si se declara una variable sin tipo de dato.

PROPIEDADES DE UN FORM



01 **(NAME)**: nombre del control para identificar en el código. Por defecto veremos "Form1".

02 **ACCEPTBUTTON**: si establecemos esta propiedad con un botón de comando, este se "activa" cuando se presiona la tecla **ENTER**.

03 **BACKCOLOR**: permite asignar el color de fondo del form.

04 **BACKGROUNDIMAGE**: permite asignar una imagen de fondo del form.

05 **CANCELBUTTON**: si establecemos esta propiedad con un botón de comando, este se "activa" cuando se presiona la tecla **ESC**.

06 **MAXIMIZEBOX**: si asignamos el valor de esta propiedad en Falso, no aparecerá en el form.

07 **MINIMIZEBOX**: si asignamos el valor de esta propiedad en Falso, no aparecerá en el form.

08 **STARTPOSITION**: indica la posición del form en la pantalla del usuario. Por ejemplo: cada vez que se ejecute el programa, la propiedad **CenterScreen** ubicará el form en el centro de la pantalla del usuario.

09 **TEXT**: en el caso del form, es para mostrar texto en la barra de títulos.

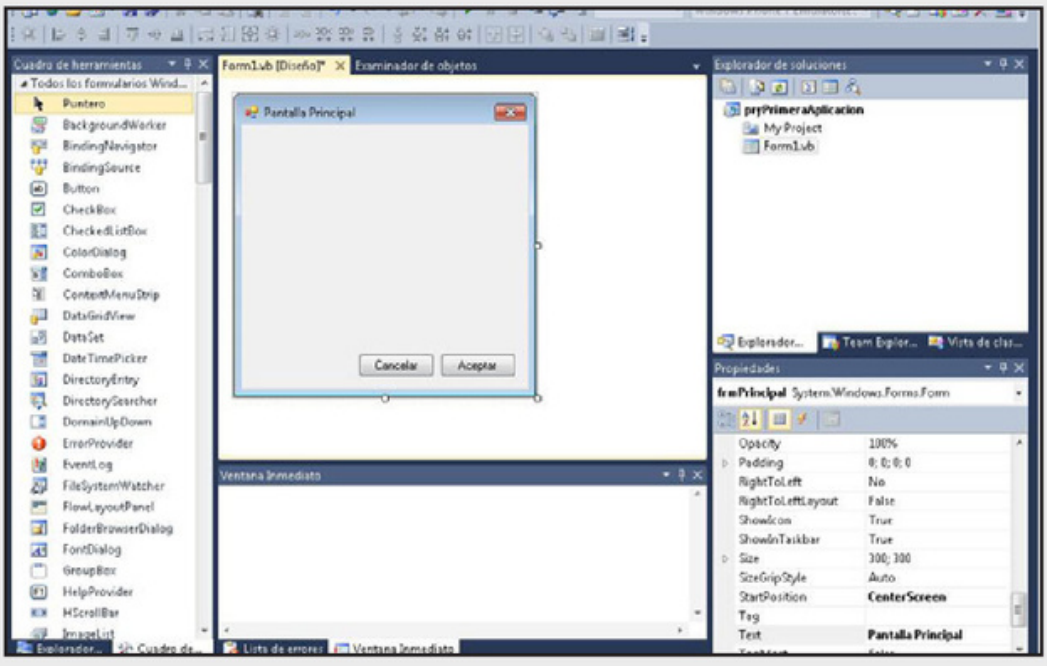
Continuando con el paso a paso anterior, ahora es el momento de desarrollar un caso práctico más complejo. La finalidad de este proyecto es que podamos comprender mejor cómo es la utilización correcta de estas herramientas y, así, poder integrar todo lo que hemos visto a través de la aplicación de las variables.

Para desarrollar este instructivo, asignemos las propiedades del **Form1** de la siguiente manera:

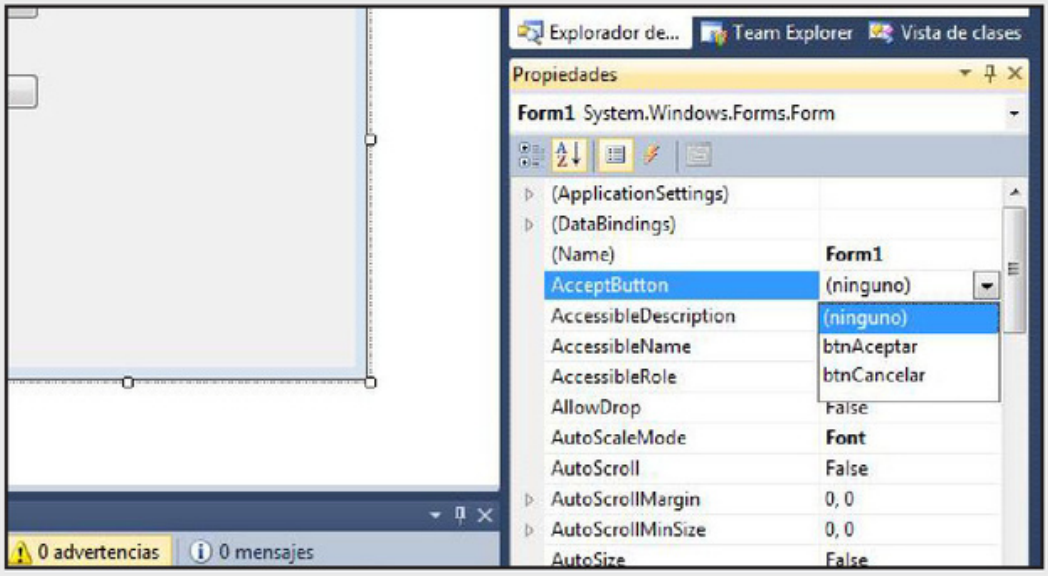
a. (Name): frmPrincipal
b. StartPosition: CenterScreen
c. Text: Pantalla Principal
d. MaximizeBox: False
e. MinimizeBox: False

PASO A PASO: COMPORTAMIENTO DE LA INTERFAZ

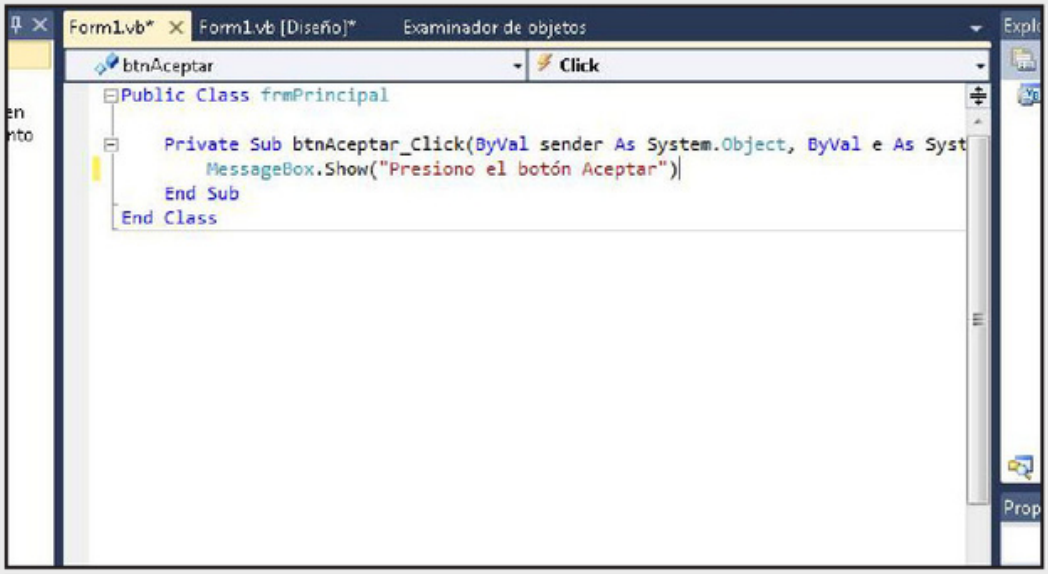
01 Para insertar dos botones de comando en el form por defecto, selecciónelos del cuadro de herramientas y utilice el botón izquierdo del mouse. Luego, asígneles las siguientes propiedades: al control **Button1**, **btnAceptar** (en **Name**) y **Aceptar** (en **Text**); y al control **Button2**, **btnCancelar** (en **Name**) y **Cancelar** (en **Text**).



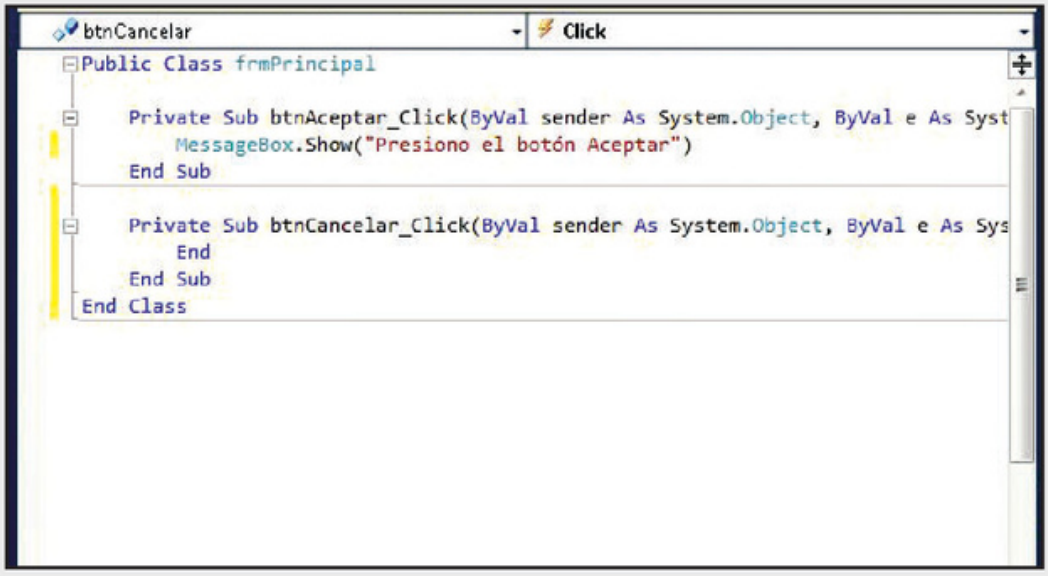
02 Luego diríjase a las propiedades del Form **AcceptButton** y asigne **btnAceptar**, y a **CancelButton: btnCancelar**.



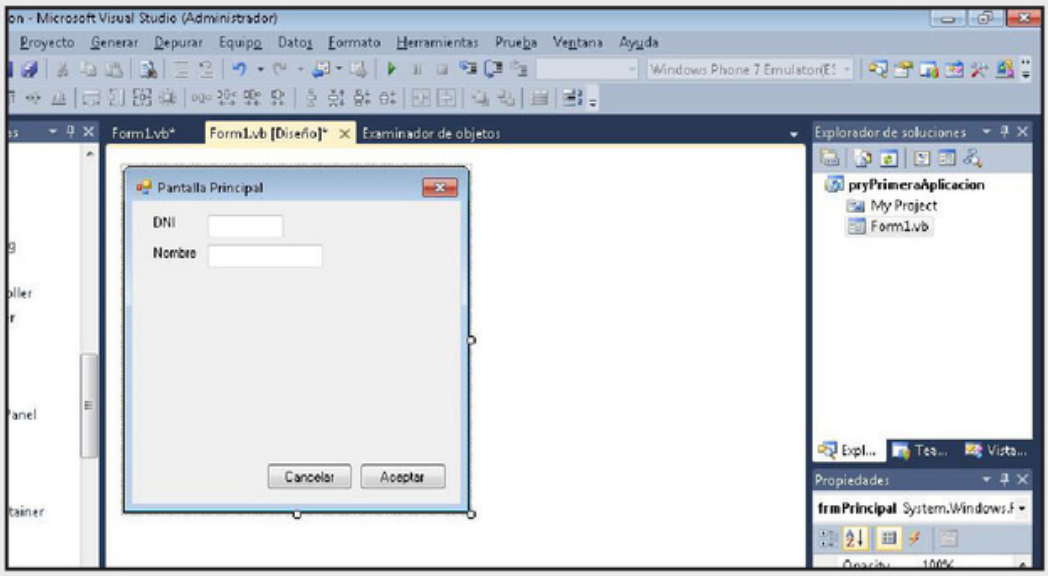
03 En el diseño visual en **btnAceptar** haga doble clic con el botón izquierdo del mouse o presione el botón **F7** para ingresar a tiempo de código. Ubíquese en el evento **Click** de **btnAceptar** y escriba el código.



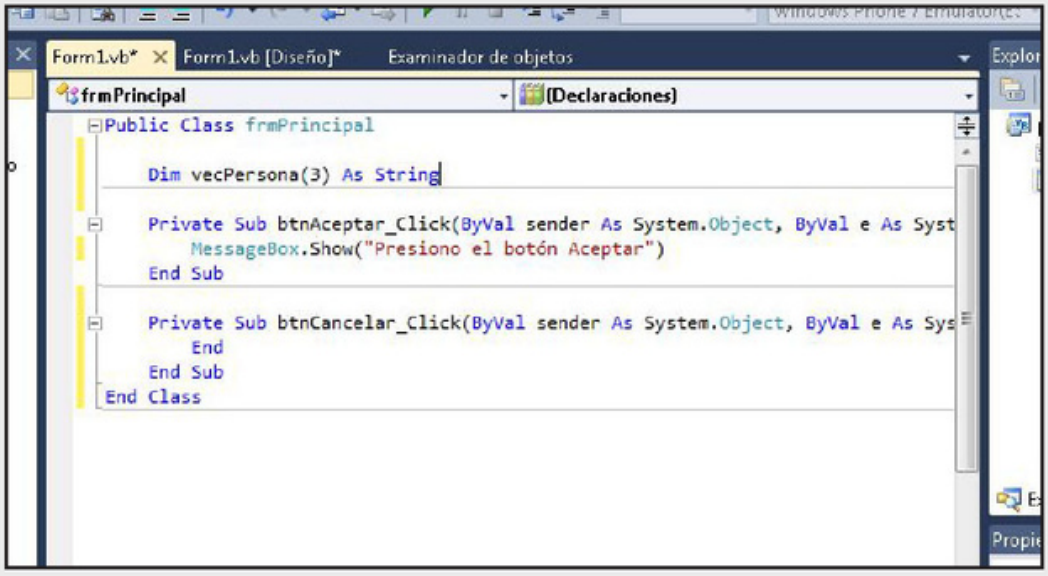
04 Ingrese en el evento **Click** del **btnCancelar** y escriba el código **End**, palabra reservada de Visual que cerrará y descargará la aplicación de memoria. Luego compile presionando **F5** o desde el menú **Depurar**, y pruebe la aplicación.



05 Para cargar datos en un vector y así grabar el DNI y nombre de una persona, dibuje dos etiquetas (**Label**) y dos cajas de texto, luego asígneles: a **Label1**, **lblDNI** (en **Name**) y **DNI** (en **Text**); a **Label2**, **lblNombre** (en **Name**) y **Nombre** (en **Text**); a **TextBox1**, **txtDNI** (en **Name**); y a **TextBox2**, **txtNombre** (en **Name**).



06 Ingrese al código del proyecto y en la zona de **Declaraciones** cree un vector de 4 espacios, de tipo cadena de texto.



Para corroborar que el usuario cargue todos los datos requeridos en los controles **TextBox**, es importante que realicemos los controles de acceso correspondientes.

- Deshabilitamos la caja de texto **txtNombre** y el botón **btnAceptar**.
- Cuando ingresemos datos en **txtDNI**, se habilitará **txtNombre**.
- Cuando ingresemos datos en **txtNombre**, se habilitará **btnAceptar**.

Veamos cómo debemos actuar para construir el código:

PROPIEDAD ENABLED

Con la propiedad **Enabled** podemos habilitar o deshabilitar controles en tiempo de ejecución. Su aplicación nos permite deshabilitar un control para restringir su uso. **True** activa el formato condicional y **False** desactiva el formato condicional. El valor predeterminado es **True**. Cuando la propiedad **Enabled** tiene el valor **True**, el formato condicional puede mostrarse en el cuadro de diálogo **Formato condicional**.
Por ejemplo, puede deshabilitarse un botón para evitar que el usuario haga clic sobre él.

a. Seleccionamos en tiempo de diseño el botón **txtNombre** en su propiedad **Enabled** igual a **False** (el control estará inhabilitado).

b. Seleccionamos el **btnAceptar** y pasamos a realizar el punto anterior.

Una vez que hayamos corroborado la carga correcta de los usuarios, hacemos doble clic con el botón izquierdo del mouse sobre **txtDNI** para acceder al evento **TextChanged**.

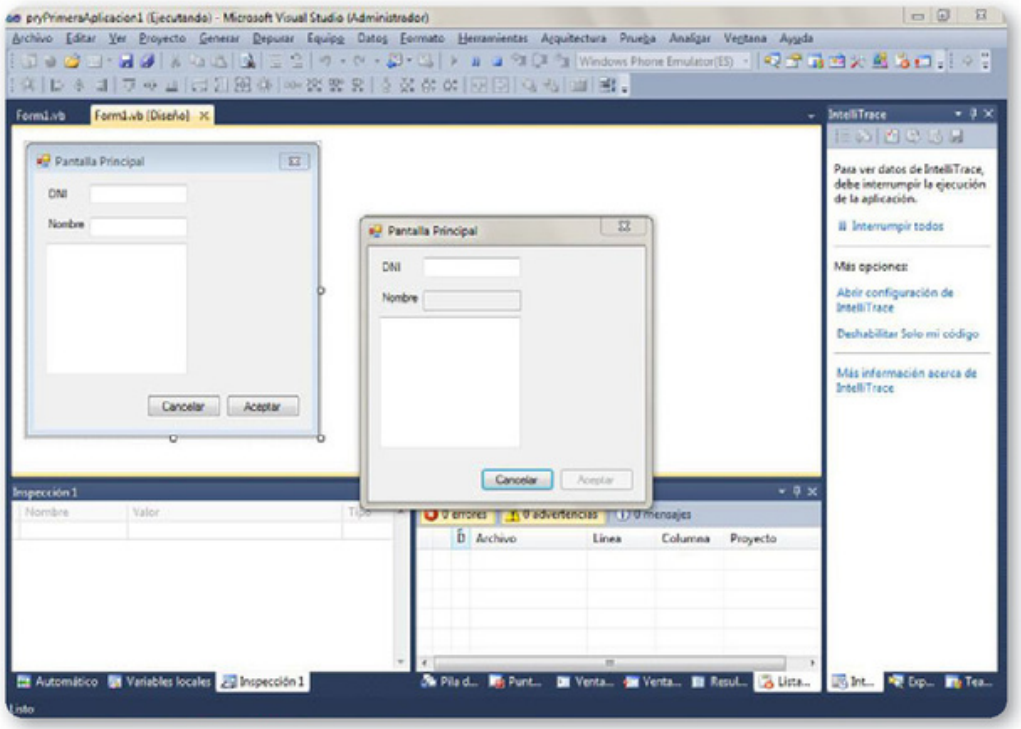


Figura 12. Controles inhabilitados que aparecen al ejecutar la aplicación y restringir el acceso inadecuado del usuario.

Para controlar el acceso de datos a la caja de texto, podemos ingresar en el evento el siguiente código:

MY

My es una de las nuevas características que presenta Visual Basic, que nos proporciona acceso a la información y a las instancias de objeto predeterminadas relacionadas con una aplicación y su entorno en tiempo de ejecución. Para conocer más: **http://msdn.microsoft.com/es-es/library/5btzf5yk.aspx**.

```
If txtDNI.Text <> Nothing Then
 txtNombre.Enabled = True
End If
```

Al ejecutar la aplicación e intentar cargar datos en DNI, notaremos que, cuando borramos la información dentro de la caja **txtDNI**, **txtNombre** quedará habilitado. Para solucionar este problema, solo debemos agregar un "sino" (**else**) en la estructura condicional:

```
If txtDNI.Text <> Nothing Then
        txtNombre.Enabled = True
    Else
        txtNombre.Enabled = False
    End If
```

Si aún quedara alguna parte que necesitemos codificar, ingresamos un dato en **txtNombre** y habilitamos el botón **btnAceptar**. De esta forma, el código final sería:

```
If txtNombre.Text <> Nothing Then
  btnAceptar.Enabled = True
    Else
        btnAceptar.Enabled = False
    End If
```

FORM

Cuando utilizamos un proyecto de Windows Form en el IDE de Visual Studio, se introduce por defecto una interfaz **FORM1**. Este **FORM** es referenciado en diferentes textos como "Forma", "Ventana" o "Formulario". Su principal objetivo es contener o agrupar controles para el uso de aplicaciones.

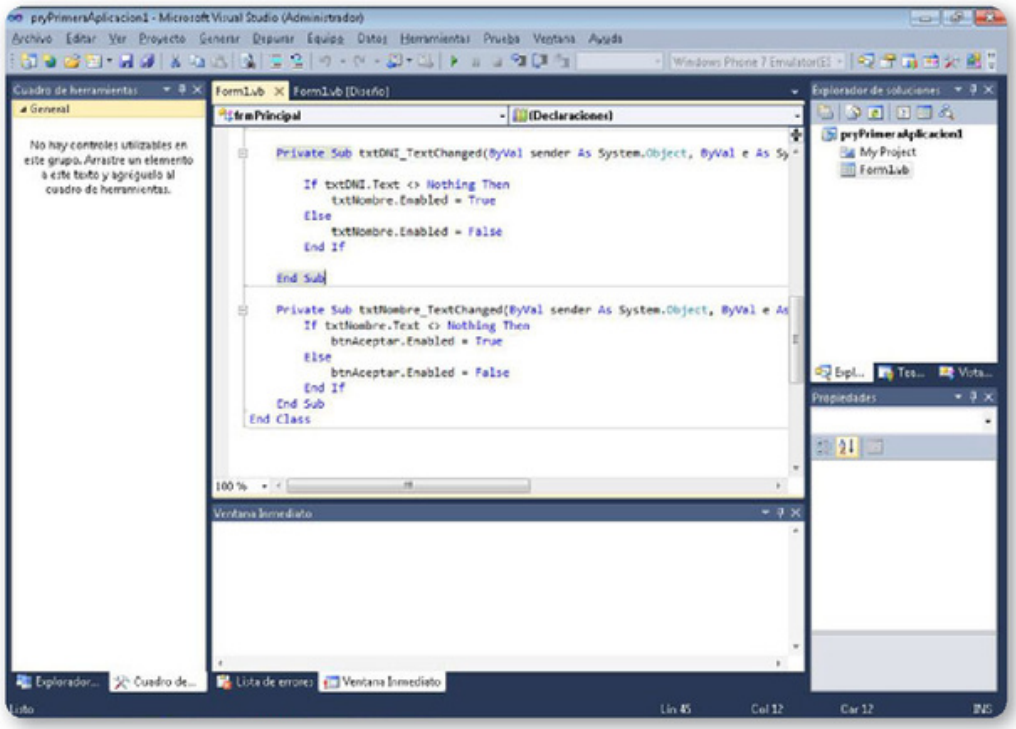


Figura 13. Código fuente referido a cómo se utiliza el evento que se ejecuta cuando ingresamos datos en una caja de texto.usuario.

Hasta aquí hemos logrado controlar el ingreso de datos sobre el form; ahora necesitaremos desplazarnos por los distintos espacios del vector que hemos creado. El código será el que se ha desarrollado en el capítulo correspondiente a matrices.

En la sección de declaraciones de variables vamos a tener que escribir lo siguiente:

```
Dim indice As Integer = 0
```

Y en **btnAceptar_Click** ingresaremos este código:

```
vecPersona(0) = txtDNI.Text & " - " & txtNombre.Text
indice = indice + 1

MessageBox.Show(vecPersona(indice))
```

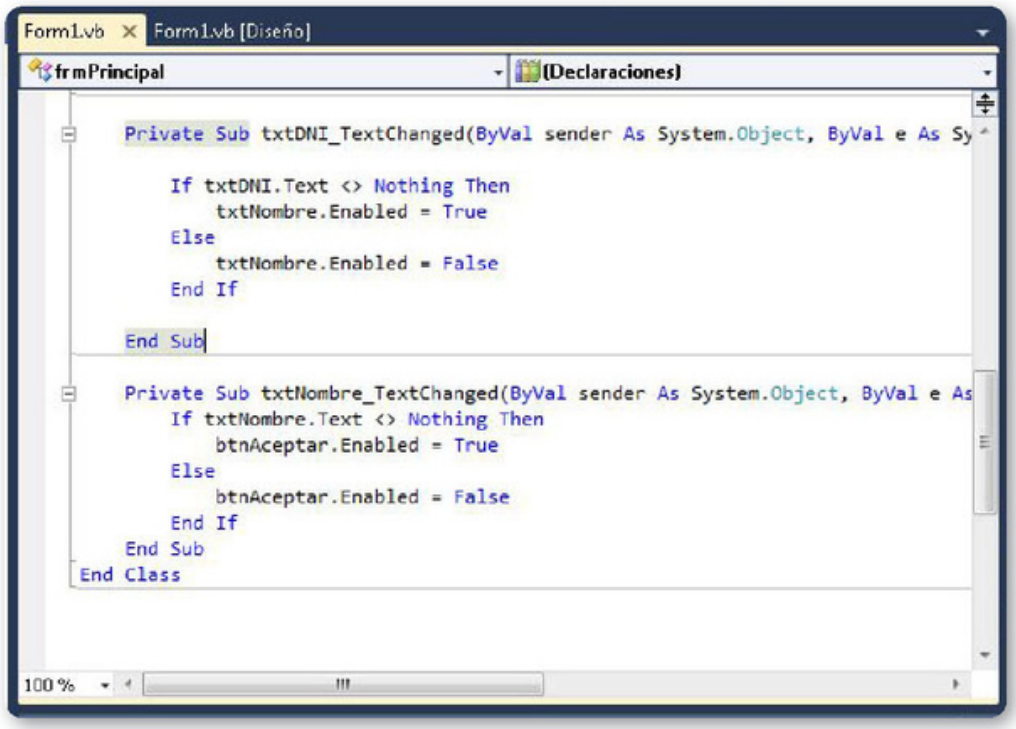


Figura 14. Código fuente del uso de matrices en Visual Basic, con las estructuras condicionales apropiadas.

De esta forma, vamos a garantizar que se carguen todos los espacios del vector. Para eso, es importante controlar la cantidad de datos ingresados; de lo contrario, aparecerá el siguiente error: "El índice está fuera de los límites del arreglo". Para que esto no suceda, debemos contar la cantidad de veces que se cargaron datos en el vector y, una vez que se completen todos, detener la aplicación e informar al usuario que ya no puede grabar más información. Por ejemplo:

```
If indice <= 3 Then
    'Cargar datos en un Vector
    vecPersona(indice) = txtDNI.Text & " - " & txtNombre.Text
    MessageBox.Show(vecPersona(indice))
    'Limpiar los controles
    txtDNI.Text = ""
    txtDNI.Focus()
    txtNombre.Text = ""
```

```
txtNombre.Enabled = False
    btnAceptar.Enabled = False
    indice = indice + 1
Else
    MessageBox.Show("Vector Completo")

    txtDNI.Text = ""
    txtDNI.Enabled = False
    txtNombre.Text = ""
    txtNombre.Enabled = False
    btnAceptar.Enabled = False
End If
```

También hay que tener en cuenta que, al cargar un dato, todos ellos quedarán cargados en el form. Por eso, es bueno "limpiar" la interfaz cada vez que entremos al evento **click** del botón. Por ejemplo:

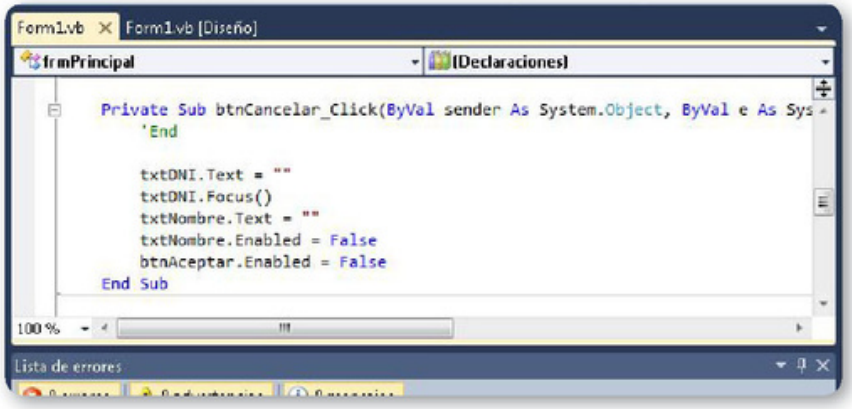


Figura 15. Código fuente donde encontramos las sentencias para limpiar los controles de la interfaz.

Ahora bien, si nos preguntamos cómo podemos mostrar la información, veremos que hay varias maneras de hacerlo. Por ahora, nosotros lo haremos agregando una **caja de texto** y asignándole a **TextBox3** las siguientes propiedades: **(Name)=txtMostrar y Multiline=True**. Es bueno tener en cuenta que la propiedad **multiline** nos permitirá ampliar el tamaño de la caja de texto en horizontal.

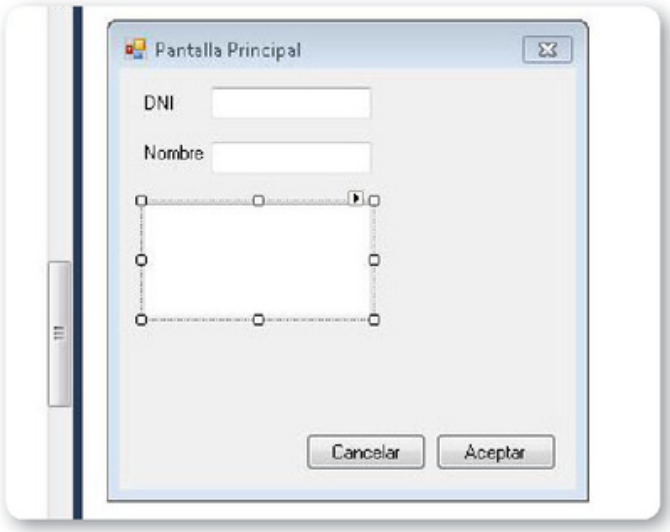


Figura 16. Interfaz gráfica donde podemos ver la inserción de un control en el que cambiamos sus propiedades a multiline.

Luego, para mostrar la información del vector, ingresamos el siguiente código:

```
txtMostrar.Text = txtMostrar.Text & " - " & vecPersona(indice) & Chr(13) & Chr(10)
```

De esta forma, hemos conocido los controles más usuales, que son el **TextBox** (caja de texto) y el **Label** (etiqueta). Además, aprendimos las propiedades para habilitar controles, cómo mostrar la información de un vector en un control y de qué manera utilizar los botones de comando en conjunto con propiedades del formulario.

RESUMEN

A lo largo del capítulo, hemos aprendido cómo se utiliza el lenguaje de programación Visual Basic para la creación de proyectos y cómo se aplican las variables en el código fuente. Luego, conocimos la manera de aplicar matrices y comparamos la sintaxis del lenguaje con pseudocódigo. También tuvimos una introducción básica a tres controles: **TextBox**, **Label** y **Button**, que utilizamos en conjunto con algunos operadores y estructuras de datos, para lograr la confección de algoritmos. Con todo lo aprendido hasta aquí, ya estamos en condiciones de comenzar a realizar los proyectos iniciales del procesamiento de datos.

Actividades

TEST DE AUTOEVALUACIÓN

1	¿Qué utilidad tiene el compilador en un lenguaje de programación?
2	¿Cuál es la utilidad de un framework?
3	¿Existen solamente lenguajes de programación de alto nivel?
4	Defina las características distintivas entre lenguajes de bajo nivel y de alto nivel.
5	¿Cómo se realiza la asignación de valores a las variables en el caso de Visual Basic?
6	¿Cuál es la forma de identificar controles en el código fuente?
7	¿Cómo es la sintaxis para declarar una variable booleana?
8	¿Cómo es la sintaxis para declarar un vector en Visual Basic?
9	¿Es correcto afirmar que en Visual Basic se declara una estructura de dato como Vector al utilizar el siguiente código: = **Dim vector(2,2) As Integer**?
10	¿Las dimensiones de las matrices bidireccionales necesitan un solo índice para recorrerlas?

*05

Primer proyecto en C++

En este capítulo continuamos avanzando en los lenguajes de programación, para introducirnos en el desarrollo de código fuente que tiene el reconocido lenguaje C++. Conoceremos un IDE gratuito para aplicar los conocimientos adquiridos anteriormente y realizaremos algunos desarrollos.

- IDE SharpDevelop..... 190
- Lenguaje de programación: C++..... 195
- Manejo de datos en C++ 203
- Cómo se utilizan los operadores..... 213
- Interactuar con el usuario 224
- Todo tiene un orden en la programación..... 230
- Datos estructurados: arrays... 243
- Resumen..... 249
- Actividades..... 250

IDE SharpDevelop

Como vimos antes, el **IDE** (entorno de desarrollo integrado) nos permitirá utilizar diferentes herramientas que faciliten el manejo de uno o varios lenguajes de programación. En el caso de **SharpDevelop**, nos encontramos con un entorno gratuito que puede utilizar los **frameworks** de Java, Microsoft y otros, y que nos permite aplicar múltiples lenguajes de programación.

Develop (abreviatura de SharpDevelop) es un entorno de desarrollo integrado libre para C #, VB.NET, Boo y proyectos en Microsoft .NET. Es de código abierto, integra distintas plantillas de códigos de programas ya creados, y se puede descargar desde www.icsharpcode.net (la versión de estas imágenes es SharpDevelop 4.2).

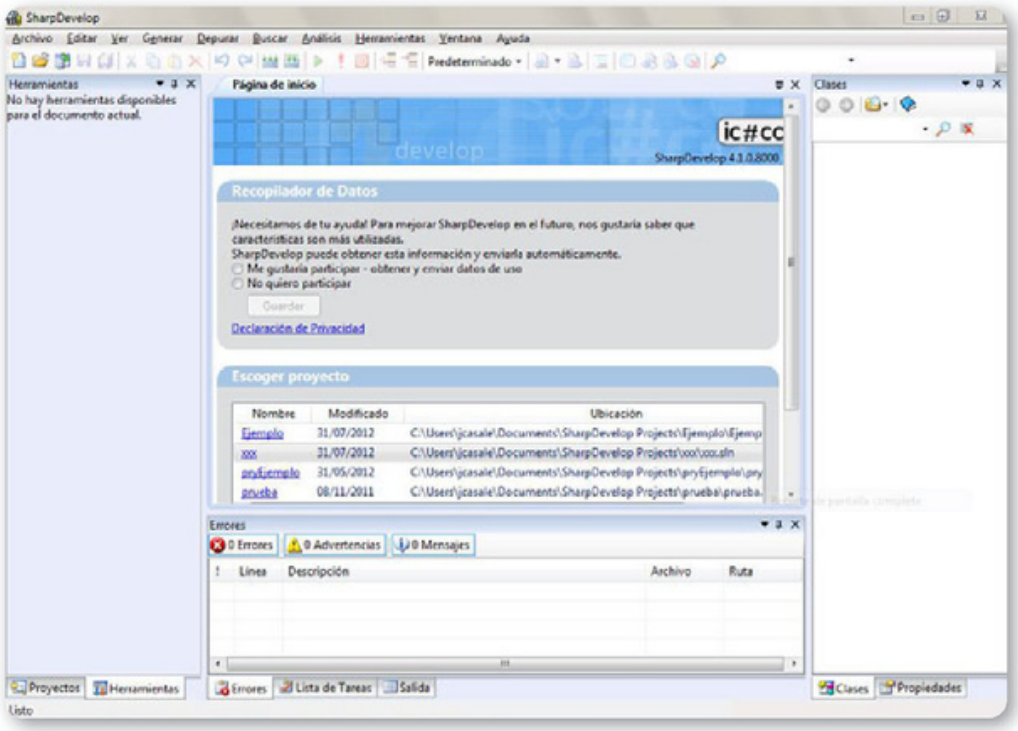


Figura 1. Interfaz gráfica del IDE, donde podemos apreciar algunas similitudes con el que utilizamos en Visual Studio.

Algunas de las características de este entorno es tener un conjunto de lenguajes variado, por ejemplo:

- Boo (finalización de código, diseñador de Windows Forms)

- C # (finalización de código, diseñador de Windows Forms)
- C++
- F #
- Python (conversión de código, diseñador de Windows Forms, completado de código parcial)
- Ruby (conversión de código, diseñador de Windows Forms)
- VB.NET (finalización de código, diseñador de Windows Forms)

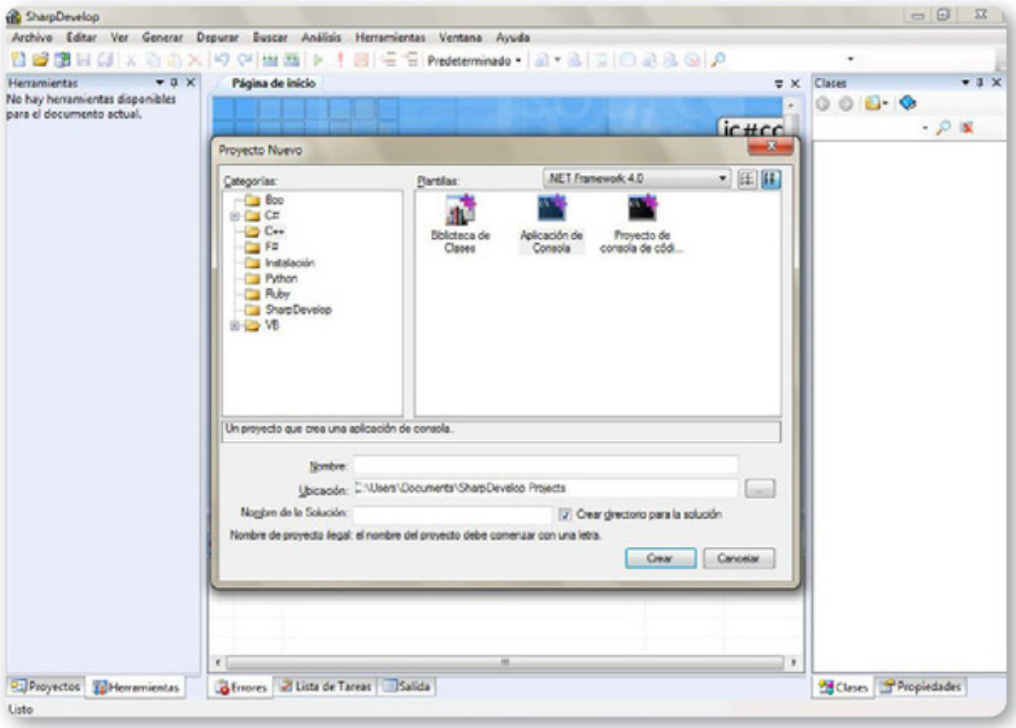


Figura 2. En esta imagen podemos ver la ventana que aparece al crear un nuevo proyecto de desarrollo en el IDE.

PYTHON

Python es un lenguaje de programación de alto nivel, cuya filosofía hace hincapié en una sintaxis muy limpia y que favorezca un código legible. Se trata de un lenguaje de programación multiparadigma, ya que soporta orientación a objetos, programación imperativa y, en menor medida, programación funcional. Es un lenguaje interpretado, usa tipado dinámico, es fuertemente tipado y multiplataforma. Podemos visitar la página oficial desde **www.python.org**.

Funcionamiento del entorno de programación

Al momento de instalar una aplicación, notaremos que por defecto la configuración del idioma estará en inglés. Sin embargo, podemos modificarla por el idioma que queramos. Para hacerlo, debemos dirigirnos al menú **TOOLS/OPTIONS** y seleccionar **Spanish**, como podemos apreciar en la **Figura 3**.

Figura 3. Con las opciones de configuración, obtenemos el entorno en español y podemos acceder fácilmente a sus funciones.

SINTAXIS EN C# Y C++

Si analizamos la sintaxis de C# y de C++, notaremos que son muy similares entre sí, ya que ambas ofrecen generalmente las mismas instrucciones de flujo de control, y la semántica es prácticamente la misma en ambos lenguajes. Sin embargo, si deseamos migrar a C#, es importante que nos informemos sobre las diferencias menores que aparecieron en su actualización.

A la hora de iniciar un desarrollo dentro de SharpDevelop, podemos crear diferentes tipos de archivos, como clases, archivos vacíos, interfases, estructura, templates, etc.

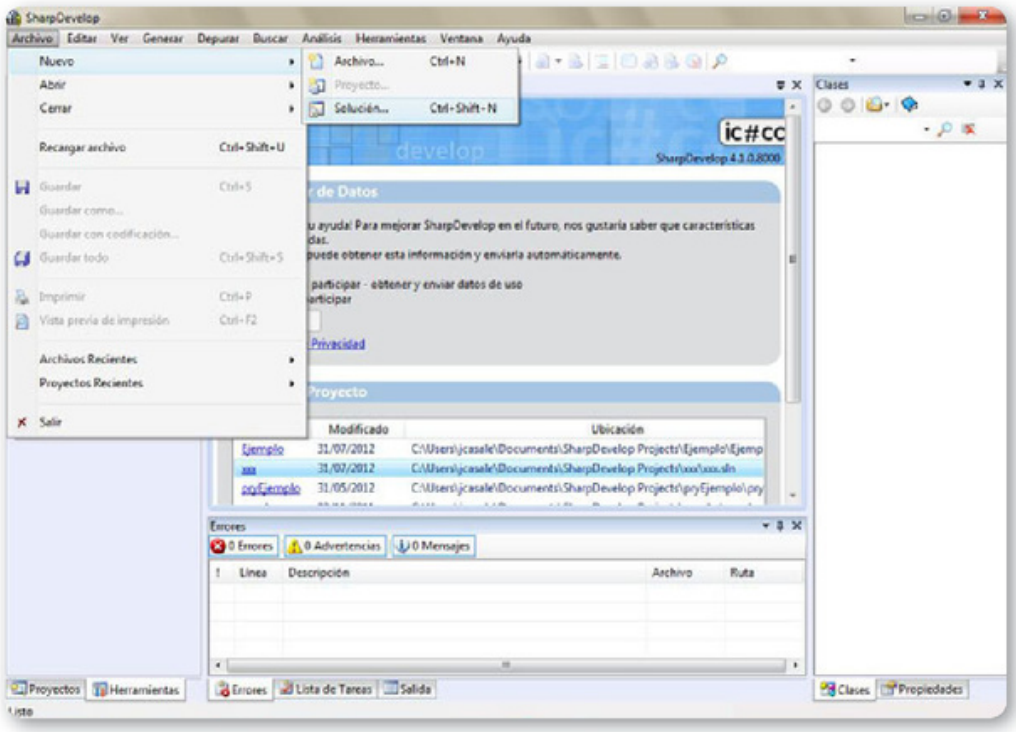


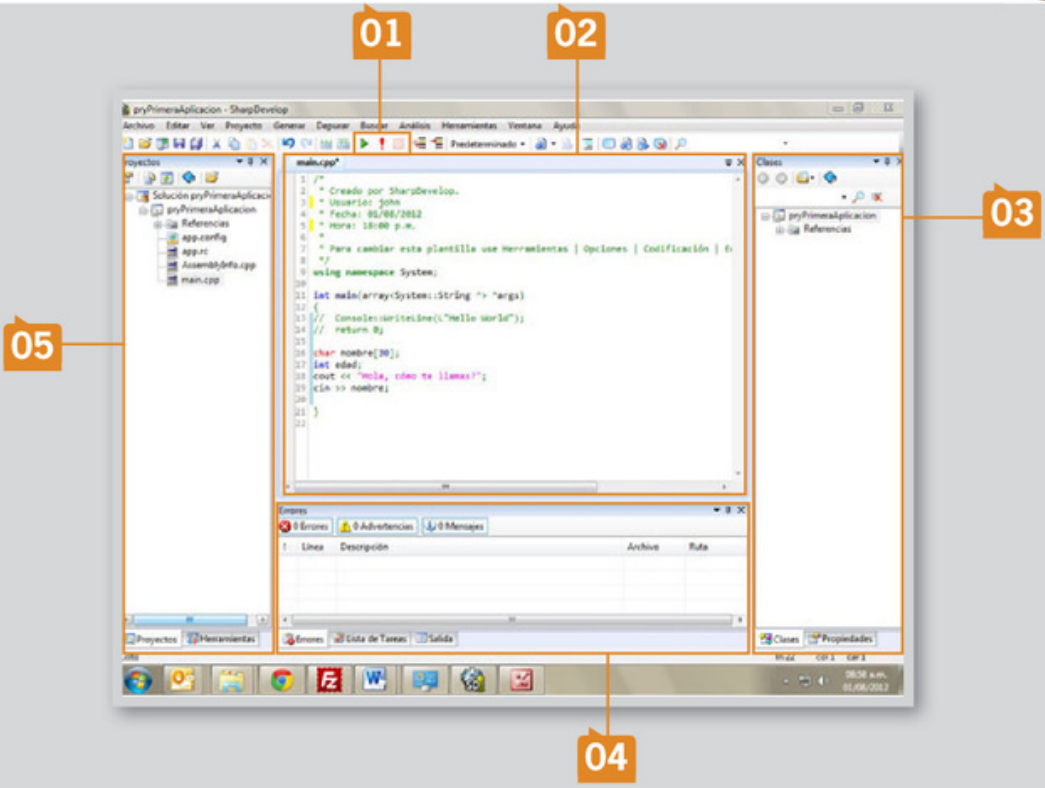
Figura 4. Desde el menú **Archivo** podemos crear diferentes archivos o soluciones.

La creación de soluciones contiene uno o varios proyectos, como utilizábamos en Visual Studio. A la hora de trabajar con SharpDevelop, primero se crea una solución y, luego, se pueden ir agregando tantos proyectos como sean necesarios.

MULTIPLATAFORMA

MonoDevelop funciona en Linux, Microsoft Windows, MacOS X, BSD, Sun Solaris, Nintendo Wii, Sony PlayStation 3 y Apple iPhone. También funciona en arquitecturas x86, x86-64, IA64, PowerPC, SPARC (32), ARM, Alpha, s390, s390x (32 y 64 bits) y muchas más. Como podemos ver, desarrollar una aplicación con Mono nos permite un gran abanico de posibilidades.

CARACTERÍSTICAS DEL IDE ■ GUÍA VISUAL 5



01 **BOTONES DE DEPURACIÓN.** Incluyen el botón **INICIO**, que compila el proyecto; **PAUSA**, que detiene la compilación o ejecución del programa; y **DETENER EL PROGRAMA**, que lo descarga de memoria.

02 **ENTORNO DE CÓDIGO.** En caso de ser proyectos con diseño, se incluye el entorno de desarrollo.

03 **CUADRO DE CLASES, PROPIEDADES, CONTENEDOR O AYUDA.** Abarca las clases involucradas en el proyecto, las propiedades que tienen los distintos elementos que vayamos utilizando y un buscador de elementos de ayuda.

04 **CUADRO DE ERRORES, TAREAS Y SALIDA DE DATOS.** Errores que pueden encontrarse en las sentencias del código fuente. Aquí también podemos configurar el seguimiento de tareas y ver las salidas de datos por medio del IDE.

05 **CUADRO DE PROYECTOS Y HERRAMIENTAS.** Aquí están los archivos involucrados en el proyecto y aquellos controles que podemos insertar dependiendo del tipo de programa que estemos diseñando.

Cada uno de estos cuadros puede configurarse como cualquier IDE, y es posible ubicarlos en diferentes lugares. La imagen que muestra la **Guía visual 1** corresponde a la manera en que veremos el entorno por defecto.

Si conocemos básicamente el uso de este entorno, comenzaremos a introducirnos en el lenguaje de programación de este capítulo. Es importante tener en cuenta que existen otros entornos gratuitos, como Mono Develop, cuya página de descarga es: http://monodevelop.com.

Lenguaje de programación: C++

Como vimos en capítulos anteriores, podemos utilizar un lenguaje de programación muy didáctico y que nos permita dibujar el diseño de las interfaces que se comunican con el usuario. En este caso, nos encontramos con un lenguaje más "duro", que no posee un entorno gráfico, sino que se caracteriza por tener un entorno de texto o CLI (*Command Line Interface,* o interfaz de comando de línea).

Antes de iniciar un programa, repasaremos algunas cuestiones importantes para tener en cuenta al utilizar C++. En principio, debemos conocer el espacio de nombres que nos permite usar sentencias de entrada y salida de datos. Por otro lado, explicaremos lo fundamental que debemos escribir (que es **main**) y veremos que C++ está orientado a una programación en eventos, aunque, actualmente, podamos adaptarlo a diferentes metodologías de desarrollo.

> ACTUALMENTE PODEMOS ADAPTAR C++ A DIFERENTES METODOLOGÍAS DE DESARROLLO

MONO DEVELOP

Se trata de un IDE diseñado principalmente para C# y otros lenguajes .NET, que permite a los desarrolladores escribir aplicaciones de escritorio y web ASP.NET en Linux, Windows y Mac OS. Facilita la existencia de una base de código único para todas las plataformas.

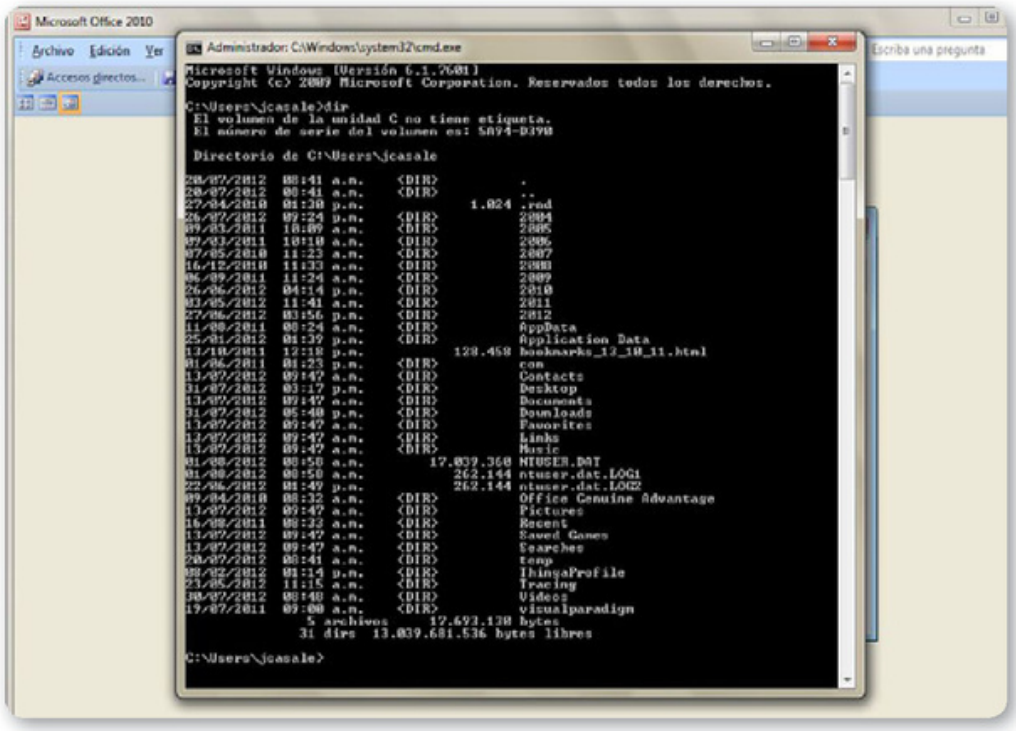


Figura 5. Interfaz de comando de línea; en este caso, entramos en la aplicación de consola de Windows.

Espacios de nombre

La instrucción **using namespace** especifica que los miembros de un **namespace** serán utilizados frecuentemente en un programa por desarrollar. Esto permite al desarrollador tener acceso a todos los miembros del namespace y escribir instrucciones más concisas. Por ejemplo, veamos a qué hacen referencia los espacios de nombre que utilizamos en C++ por defecto y cómo sería si no lo usáramos.

ORIGEN DE C++

Es un lenguaje de programación diseñado a mediados de los años 1980 por Bjarne Stroustrup, cuya intención fue extender el exitoso **lenguaje de programación C** con mecanismos que permitieran la manipulación de objetos. En ese sentido, desde el punto de vista de los lenguajes orientados a objetos, el C++ es considerado un lenguaje híbrido. El nombre C++ fue propuesto por Rick Mascitti en el año 1983.

```
#include <iostream>
int main ()
{
std::cout <<"hola mundo"<<std::endl;
return 0;
}
Sin utilizar el espacio de nombre std

#include <iostream>
using namespace std;
int main ()
{
//al usar el namespace ya no hace falta escribir std::
cout <<"hola mundo"<<endl;
return 0;
}
Utilizando el espacio de nombre std
```

A continuación, veremos qué quiere decir cada sintaxis del código que escribimos anteriormente. La idea principal es conocer cómo se escribiría el código en C++ utilizando espacios de nombres que simplificarán la codificación de manera notable.

Nosotros podemos codificar nuestros propios espacios de nombres, por ejemplo: en C++ crearemos uno llamado "calcular" y le agregaremos variables **varX**, **varY** y **varF**. Veamos el código:

```
namespace calcular {
  int varX, varY;
float varF(int a) {
    // ...
  }
  }
```

Con este espacio de nombre, para utilizar la variable "**varX**" que está en el namespace "calcular", el desarrollador deberá codificar lo siguiente:

```
calcular::varX = calcular::varF(5);
```

Si utilizáramos de esta forma la sintaxis, deberíamos escribir reiteradas veces la palabra "calcular". En cambio, si usamos el espacio de nombre, quedaría de la siguiente forma:

```
using namespace calcular;

varX = varF(5);
```

Veamos cómo desarrollamos la codificación si usamos dos espacios de nombres:

```
// namespaces
#include <iostream>
using namespace std;

namespace primero
{
  int var = 5;
}

namespace segundo
{
  double var = 3.1416;
}
```

ESPACIO DE NOMBRE

Así como en Visual Basic podemos utilizar namespace, el contenido de este nos permite agrupar o utilizar un conjunto de clases a nivel global, los objetos y / o funciones que le confieren un nombre. Por ejemplo, el espacio de nombre que se debe utilizar en C++ es **std**, con la sintaxis **using**.

```
int main () {
  cout << primero::var << endl;
  cout << segundo::var << endl;
  return 0;
}
```

Creando dos espacios de nombre

Como podemos observar, cada espacio de nombre es propietario de su variable, y en la codificación podemos utilizar estas como queramos.

Conceptos básicos del código

Es recomendable repasar la **Tabla 1**, donde encontraremos la estructura básica que podemos realizar en el código de C++. De esta forma, podremos comprender o manejar mejor aquellas palabras y signos que nos facilitan la programación.

ESTRUCTURA

TIPO	DESCRIPCIÓN
//	Representa en C++ los comentarios que podemos agregar en el lenguaje de programación, que no tienen ningún efecto sobre el comportamiento del programa.
#include	Las líneas que comienzan con un numeral (#) son directivas para el preprocesador. En este caso, la directiva #include indica al preprocesador incluir el archivo iostream estándar.
Iostream	Este archivo específico incluye las declaraciones de la norma básica de la biblioteca de entrada-salida en C ++, y se ha incluido debido a que su funcionalidad será utilizada más adelante en el programa.
using namespace std;	Todos los elementos básicos de las bibliotecas en C++ se declaran dentro de lo que se llama un espacio de nombres. Por eso, si queremos acceder a su funcionalidad, declaramos con esta expresión para poder utilizarla. Esta línea es muy frecuente en C++ para que los programas utilicen las bibliotecas estándar.
int main ()	Esta línea corresponde al comienzo de la definición de la función principal. Se trata del punto por el cual todos los programas en C++ comienzan su ejecución, independientemente de su ubicación dentro del código fuente.

Cout<<"Usando C++!";	Cout es la secuencia de salida estándar en C++. El significado de toda la declaración es insertar una secuencia de caracteres (en este caso, "Usando C++!"). Cout se declara en iostream, que es el estándar de archivos en el espacio de nombres. Por eso, debemos incluir esa referencia específica y declarar que vamos a utilizar ese espacio de nombres a principio del código. Es importante no olvidar el punto y coma (;) que aparece al final de la sentencia.
;	Este carácter se utiliza para marcar el final de la declaración, y debe incluirse al término de todas las declaraciones de expresión de todos los programas en C++ (uno de los errores de sintaxis más comunes es olvidarnos de incluir un punto y coma después de una declaración).
Return 0;	La sentencia return hace que la función principal llegue a su fin, y puede estar seguida por un código (el código de retorno devuelve un valor cero). En el código de retorno de 0 para el main de la función se interpreta que el programa funcionó como se esperaba, sin ningún error durante su ejecución. Esta es la manera más habitual de poner fin a un programa en C++.

Tabla 1. En esta tabla podemos ver la estructura básica de C++ que utilizaremos en nuestros proyectos de programación.

Inmediatamente después de estos paréntesis, se encuentra el cuerpo de la función principal que va a ir encerrada entre llaves {}. Esto es lo que hace la función al momento de ejecutarse.

Ahora que conocemos en detalle la estructura de la programación en C++, podemos reconocer cuáles son las diferentes sentencias y qué sectores serán importantes en el desarrollo del lenguaje. En los próximos párrafos, iremos viendo la aplicación de nuestro primer desarrollo en C++ con el IDE SharpDevelop.

NAMESPACE STD

En C++ hay código estándar en el namespace **std**. Podemos utilizarlo nombrándolo en el encabezado del proyecto de la siguiente forma: **using namespace std**, o bien podemos utilizar en nuestro código constantemente **std::** y realizar las llamadas a las funciones que necesitemos. Todas las librerías estándar de C++, formadas por la ISO para implementar operaciones comunes, están incorporadas en **std**.

Primera aplicación en C++

En esta sección vamos a empezar a utilizar el entorno de SharpDevelop y crear nuestra primera aplicación en C++ para consola (CLI). Para eso, vamos a ejecutar SharpDevelop.

Desde el menú **Archivo**, hacemos clic en **Nuevo** y seleccionamos **Solución**. Cuando aparece el cuadro de diálogo **Nuevo proyecto**, seleccionamos el lenguaje de programación C++ y, en el cuadro contiguo, **Aplicación de consola**. En el cuadro **nombre** escribimos **pryPrimeraAplicacion**. A continuación, veremos una interfaz gráfica parecida a la de la **Figura 5**. Veamos un ejemplo en el siguiente código:

```
/*
 * Creado por SharpDevelop.
 *
 * Para cambiar esta plantilla use Herramientas | Opciones | Codificación |
    Editar Encabezados Estándar
 */
using namespace System;

int main(array<System::String ^> ^args)
{
 Console::WriteLine(L"Hello World");
 return 0;
}
Creando dos espacios de nombre
```

Para seguir en nuestro proyecto, debemos compilar este código presionando **F5** o hacer clic sobre el botón **Inicio de depuración**;

MAIN EN C++

La palabra **main** es seguida en el código por un par de paréntesis **()**. Esto es así porque se trata de una declaración de la función: en C++, lo que diferencia a una declaración de la función de otros tipos de expresiones son estos paréntesis que siguen a su nombre. Opcionalmente, estos podrán adjuntar una lista de **parámetros** dentro de ellos.

veremos como resultado una ventana de consola que muestra "Hello Word" y se cierra. Luego modificamos este código, agregando el espacio de nombre **std**, para poder ingresar y mostrar datos, y también la librería **iostream**.

```
using namespace std;
#include <iostream>

Ahora vamos a utilizar las estructuras básicas que vimos antes para ver el
    mensaje en pantalla, y reemplazamos el código que hay en main por el
        siguiente.
// Mostramos por salida estándar un mensaje personalizado
cout << "Mi primera aplicación en C++!\n";
return 0;
```

Para poder apreciar la aplicación, agregaremos una palabra reservada de C++ que es **getchar()**, de modo que, al presionar la tecla **ENTER**, cerrará la ventana de consola. El código es:

```
getchar();
```

Hasta aquí hemos visto cómo utilizar el entorno de SharpDevelop y crear nuestra primera aplicación en C++ para consola, desde el IDE que estamos viendo en este capítulo. A continuación, vamos a seguir trabajando con el mismo ejemplo, para reforzar el concepto.

Manejo de datos en C++

Continuaremos con nuestra primera aplicación de C++, pero esta vez agregaremos la declaración de variables y muestra de información en ventana, y veremos qué tipos de datos podemos utilizar.

Como ya hemos visto, al crear una variable, disponemos de un espacio de memoria donde podemos almacenar cierta información, pero debemos indicar qué tipo de información será. Por ejemplo, almacenar una letra, un número entero o un booleano.

Tipos de datos

Veamos en la **Tabla 2** cuáles son los tipos de datos fundamentales en C++ y el rango de valores que se pueden representar.

DATOS

NOMBRE	DESCRIPCIÓN	TAMAÑO	RANGO
Char	Carácter o entero pequeño	1 byte	signed: -128 to 127 unsigned: 0 to 255
short **int(short)**	Entero corto	2 bytes	signed: -32768 to 32767 unsigned: 0 to 65535
Int	Entero	4 bytes	signed: -2147483648 to 2147483647 unsigned: 0 to 4294967295
long int **(long)**	Entero largo	4 bytes	signed: -2147483648 to 2147483647 unsigned: 0 to 4294967295

SENSIBILIDAD

El lenguaje C++ es "case sensitive". Esto significa que un identificador escrito en letras mayúsculas no es equivalente a otro con el mismo nombre escrito en minúsculas. Así, por ejemplo, la variable **RESULTADO** no será lo mismo que **resultado** o **Resultado**. Se trata de tres identificadores de variables diferentes. El caso contrario sería case insensitive, en donde el uso de variables es totalmente indiferente.

Bool	Boolean. Puede adquirir uno de dos valores: true o false	1 byte	true o false
Float	Punto flotante	4 bytes	+/- 3.4e +/- 38 (~7 dígitos)
Doublé	Doble precisión de punto flotante	8 bytes	+/- 1.7e +/- 308 (~15 dígitos)
long doublé	Doble largo de precisión de punto flotante	8 bytes	+/- 1.7e +/- 308 (~15 dígitos)
wchar_t	Carácter amplio	2 o 4 bytes	1 carácter amplio

Tabla 2. En esta tabla podemos ver los diferentes tipos de datos en C++ que podemos utilizar en el desarrollo de aplicaciones.

Los valores de las columnas Tamaño y Rango van a depender del sistema en donde se compile el programa, y son los que se encuentran en la mayoría de los sistemas de 32 bits. A continuación, desarrollaremos la declaración de variables necesaria para aplicar a nuestro primer proyecto de C++.

Declaración de variables

Para utilizar una variable en C++, primero se debe declarar qué tipo de datos deseamos y, luego, sus respectivos nombres. En la sintaxis para declarar una nueva variable se escribe la especificación del tipo de datos (**int**, **bool**, **float**, etc.), seguido de un identificador de variable válido. Por ejemplo:

```
int varX;
float varP;
char caracter;
char cadena[10];
```

También podemos realizar, como en Visual Basic, la declaración consecutiva de variables:

```
int varX, varY, varZ;
```

Veremos que en C++ podemos asignar o no signos positivos o negativos en la declaración de variables. Los tipos de datos **char**, **short**, **long** e **int** pueden ser con o sin signo (positivo o negativo) en función del rango de números necesarios para ser representado. Las declaraciones con signo pueden representar los valores tanto positivos como negativos, mientras que los tipos sin signo solo representan valores positivos hasta el cero. Por ejemplo:

```
unsigned short int varNumeroDeVeces;
signed int varMiSaldo;
```

En caso de que declaremos una variable sin especificar **signed** o **unsigned**, C++ por defecto especificará la primera. Por lo tanto, en el caso de la variable **varMiSaldo**, podríamos haberlo hecho de la siguiente forma:

```
int varMiSaldo;
```

Continuando con nuestra primera aplicación en C++, vamos a utilizar algunas variables que nos permitan realizar una tarea matemática y mostrarlo en pantalla:

```
using namespace std;
#include <iostream>
```

ENMASCARAMIENTO DE VARIABLES

Por lo general resulta difícil o innecesario declarar dos variables con el mismo nombre. Sin embargo, hay condiciones bajo las cuales es posible hacerlo. Para eso, se puede declarar una variable global con un nombre determinado y declarar otra (del mismo tipo o de otro diferente) de forma local en una función, usando el mismo nombre.

```
int main()
{
  // declarar variables:
  int varA, varB;
  int resultado;

  // instrucciones:
  varA = 7;
  varB = 2;
  varA = varA + 1;
  resultado = varA - varB;

  // mostramos el resultado:
  cout << resultado;

  // terminamos el programa:
  getchar();
  return 0;

}
```

De esta forma, veremos el resultado en una ventana de consola, que es 6. Al igual que en Visual Basic, las variables tienen un ámbito donde pueden operar: serán globales o locales dependiendo del lugar en donde se declaren. Mirando el gráfico de la **Figura 6** podremos visualizar claramente este concepto.

Como vimos antes, las variables globales pueden llamarse desde cualquier lugar del proyecto, mientras que las locales solo pueden llamarse cuando los bloques de código estén comprendidos por llaves, {}.

VARIABLES EN C# Y C++

Si analizamos las definiciones de variables de C# y de C++ notaremos que son muy similares entre sí. En ambos lenguajes las variables pueden ser declaradas solo localmente en un método o como miembros de una clase. En el caso de C#, no se ofrecen equivalentes a las variables globales o estáticas de C++.

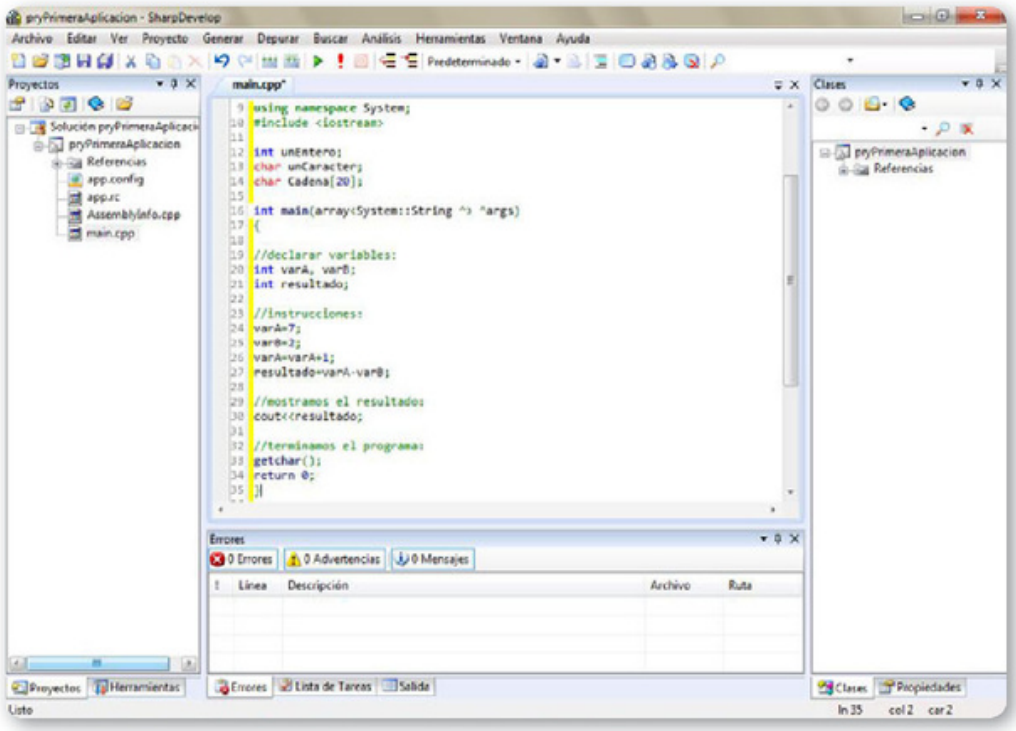


Figura 6. En esta imagen podemos observar el ámbito de las variables que son declaradas antes del **`main`**.

Inicializar variables

A la hora de declarar las variables locales, es importante tener en cuenta que si no asignamos un valor al momento de crearlas, estas tendrán un valor indeterminado.

Desde C++, veremos dos maneras posibles de almacenar un valor concreto al momento de declararlas:

- La primera, conocida como inicialización **c-like**, se logra añadiendo un signo igual (=) seguido del valor que deseamos inicializar:

identificador de tipo de dato = valor inicial;

Por ejemplo, si queremos declarar una variable **int** inicializado con el valor 0 en el momento que lo declaramos, podríamos escribir:

```
int varA = 0;
```

- La segunda implica inicializar las variables como **constructor de inicialización** y encerrar el valor inicial entre paréntesis, **()**.

Identificador de tipo de dato (valor inicial);

Por ejemplo:

```
int vara (0);
```

Ambas formas de inicialización son válidas en C++. Teniéndolas en cuenta, ahora las aplicaremos en el ejemplo anterior de nuestro primer proyecto. De esta forma podremos ahorrarnos un par de líneas de código y así agilizar nuestro trabajo.

```
using namespace std;
#include <iostream>

int main()
{
 // declarar variables:
 int varA = 7;
 int varB(2);
 int resultado;

 // instrucciones:
 varA = varA + 1;
 resultado = varA - varB;

 // mostramos el resultado:
 cout << resultado;

 // terminamos el programa:
 getchar();
 return 0;

}
```

Para practicar en C++ lo que vimos en estos párrafos, vamos a codificar un nuevo ejemplo que muestre el tamaño en bytes de cada tipo de dato. Una vez ejecutado el SharpDevelop, desde el menú **Archivo**, hacemos clic en **Nuevo** y seleccionamos **Solución**. Aparecerá el cuadro de diálogo **Nuevo proyecto**, donde seleccionamos el lenguaje de programación C++ y, en el cuadro contiguo, **Aplicación de consola**. En **nombre** escribimos **pryTipoDeDatos**. A continuación, en el cuadro de código realizamos lo siguiente:

```
using namespace std;
#include <iostream>

int main()
{

// Sacamos el tamaño de cada tipo
cout << "El tamaño del int es:\t\t" << sizeof(int) << " bytes.\n";
cout << "El tamaño del short es:\t" << sizeof(short) << " bytes.\n";
cout << "El tamaño del long es:\t" << sizeof(long) << " bytes.\n";
cout << "El tamaño del char es:\t\t" << sizeof(char) << " bytes.\n";
cout << "El tamaño del float es:\t\t" << sizeof(float) << " bytes.\n";
cout << "El tamaño del double es:\t" << sizeof(double) << " bytes.\n";
// Sacamos por salida estándar un mensaje
cout << "Termino el programa\n";

  getchar();
  return 0;

}
```

SIGNIFICADOS EN C# Y C++

Si analizamos los significados y sintaxis de C# y de C++ notaremos que son muy similares entre sí. Los operadores que tiene por defecto C# repesentan la misma sintaxis y semántica que en C++. Si bien el uso de **()**, **[]** y **,** (comas) cumple el mismo efecto, debemos tener cuidado con: **Asignación** (=), **new** y **this**.

Si queremos ver el resultado final, podemos hacerlo compilando con **F5** o haciendo clic en **Iniciar depuración**.

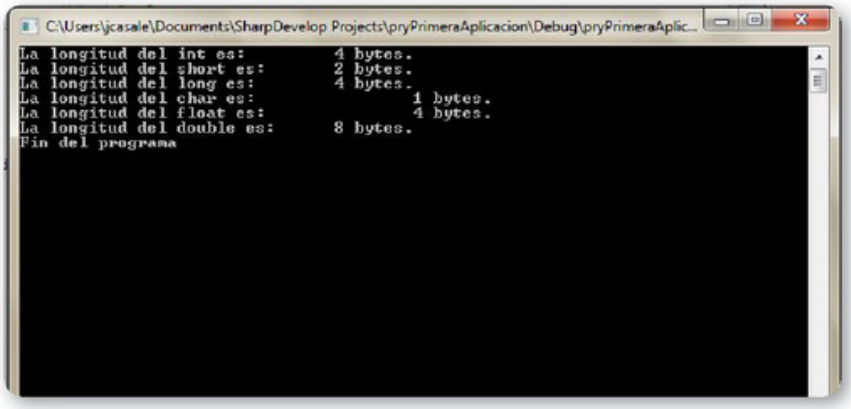


Figura 7. Resultado del ejemplo que codificamos antes, mostrando el espacio que ocupa cada tipo de dato.

Cabe destacar que existen ciertas palabras reservadas, como **sizeoff**, **cout** y **bytes**, que provienen de las librerías base de C++. Más adelante veremos qué uso darle a cada una de ellas.

Además del uso de variables y la asignación de su tipo de dato, también podemos declarar constantes si agregamos al proyecto anterior las siguientes líneas de código, antes del **getchar()**:

```
using namespace std;
#include <iostream>

#define FINAL 25

int main()
{
//... todas las instrucciones anteriores
int y = 0;

// Definimos el valor constante
const float PI = 3.1416;
cout << "La constante PI contiene: " << PI << endl;
```

```
// Sacamos por salida estándar un mensaje
cout << "\nTermino el programa : " << FINAL << endl;
  getchar();
  return 0;
}
```

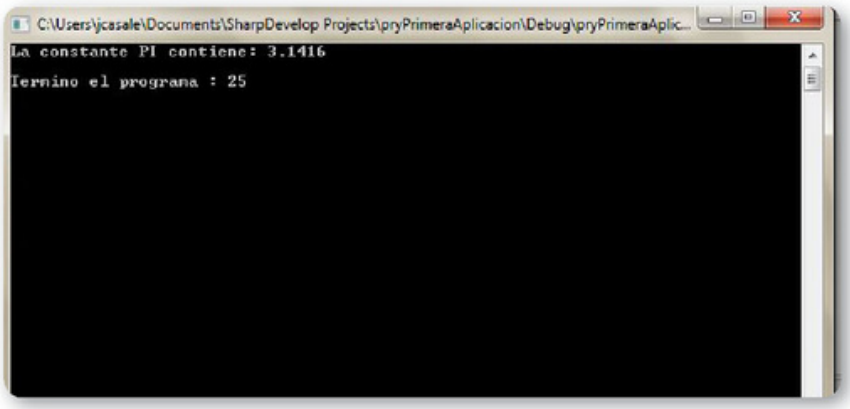


Figura 8. Resultado del ejemplo donde estamos utilizando una constante con punto flotante.

Podemos observar que, a la hora de declarar, existen dos posibilidades: una antes del **main** con la palabra **#define**, y otra con la palabra **const**. A continuación, vamos a trabajar con mayor detenimiento en cada una de estas declaraciones.

Formas de declarar constantes

En C++ las constantes se declaran y no van precedidas por ninguna palabra reservada que indique la sección.

Literales

Son las constantes más frecuentes y se utilizan para expresar valores particulares dentro del código fuente de un programa. Las hemos utilizado previamente cuando les dábamos valores a las variables o cuando expresábamos los mensajes que queríamos mostrar en los programas. Por ejemplo, cuando escribimos:

```
varA = 3;
```

El 3 en esta instrucción de código fue una constante literal. Es bueno tener en cuenta que las constantes literales pueden dividirse en: números enteros, números de punto ⊠otante, caracteres, cadenas de texto y valores booleanos.

Constantes definidas (# define)

En este lenguaje podemos declarar nombres de constantes con el comando **#define**, sin tener que declararlas dentro de una función. Podemos utilizarlas en el encabezado del programa y llamarlas desde cualquier proceso.

Su sintaxis es la siguiente:

```
#define valor del identificador
```

Por ejemplo:

```
#define PI 3.14159
#define cabecera "Empresa";
```

Constantes declarados (const)

Al igual que vimos en Visual Basic, encontraremos que la sintaxis **const** representa la declaración de constantes con un tipo específico, de la misma manera en que lo haría con una variable:

```
const int varCantHojas = 100;
const char varLetra = 'x';
```

Como vimos en capítulos anteriores, podemos tratar a las constantes del mismo modo que lo hacíamos con las variables normales. La única diferencia se encuentra en sus valores, ya que estos no podrán ser modificados luego de su definición.

Cómo se utilizan los operadores

Si bien hasta el momento vimos cómo se utilizan los operadores en Visual Basic, ahora podremos practicarlos en C++ considerando su importante diferencia en cuanto a codificación y manejo de librerías. A continuación, haremos un recorrido por los distintos operadores que encontramos en C++ y ejemplos prácticos que nos permitan conocer mejor su funcionamiento.

Asignación (=)

El operador de asignación se ocupa de cargar un dato a una variable. Si quisiéramos establecer que la variable A contenga el valor 3, su sentencia de código sería:

```
varA = 3;
```

La parte de la izquierda del operador de asignación (=) se conoce como el **lvalue** (valor de la izquierda), y la derecha, como el **rvalue** (valor derecho). El valor izquierdo tiene que ser una variable, mientras que el derecho puede ser una constante, una variable, el resultado de una operación o cualquier combinación de ellos.

Debemos recordar que las asignaciones se realizan de derecha a izquierda, y nunca al revés. Veamos el siguiente ejemplo:

```
int varA, varB;        / / varA: , varB:
 varA = 10;          / / varA: 10, varB:
 varB = 4;          / / varA: 10, varB: 4
 varA = varB;          / / varA: 4, varB: 4
 varB = 7;          / / varA: 4, varB: 7
```

En el código anterior, el resultado puede mostrar que **varA** es igual a 4, y **varB** es igual a 7. Notemos que las variables no se vieron afectadas por la modificación al final de **varB**, a pesar de que se declaró **varA =**

varB, es decir, debido a la regla de la asignación de derecha a izquierda.

De pronto, si buscamos por la Web algunos ejemplos, podremos encontrarnos con una asignación del siguiente tipo:

```
varA = varB = varC = 7;
```

Operadores aritméticos

Ya hemos usado la mayoría de los operadores en otro lenguaje y, en el caso de C++, notaremos que su forma de aplicación no cambia. Revisemos en la **Tabla 3** cuáles son los operadores que podemos utilizar.

ARITMÉTICOS

SIGNO	OPERACIÓN
+	Suma
-	Resta
*	Multiplicación
/	División
%	Módulo

Tabla 3. Aquí podemos observar los distintos operadores aritméticos que utilizaremos para realizar operaciones matemáticas o lógicas.

Antes de pasar a algunos ejemplos, veamos de qué se trata el operador módulo (%). Cuando hablamos de **módulo** nos referimos a la operación que presenta el resultado de una división de dos valores. Por ejemplo, si escribimos:

```
int varA = 11 % 3;
```

En este caso, el resultado que obtendremos es 2, ya que 2 es el residuo obtenido de la división de 11 en 3.

Asignación compuesta

Cuando queremos modificar el valor de una variable mediante la realización de una operación, podemos utilizar operadores de asignación compuestos. Veamos la **Tabla 4**, que detalla cada uno de ellos.

COMPUESTA

EXPRESIÓN	ES EQUIVALENTE A
valor += incremento;	valor = valor + incremento;
a -= 5;	a = a - 5;
a /= b;	a = a / b;
precio *= unidad + 1;	precio = precio * (unidad + 1);

Tabla 4. En esta tabla podemos observar los operadores de asignación compuesta, propios del lenguaje de programación.

También podemos encontrarnos con más operadores, tales como: **+=**, **-=**, ***=**, **/=**, **%=**, **>>=**, **<<=**, **&=**, **^=** y **|=**.

A continuación, veamos un modelo en código en el cual debemos generar una aplicación de ejemplo y codificar lo siguiente:

```
        using namespace std;
#include <iostream>

int main()
{
          int varA, varB=3;
          varA = varB;
          varA+=2;          // equivale a=a+2
          cout << varA;
          getchar();
          return 0;
        }
```

Aumentar y disminuir

En varios lenguajes de programación encontraremos signos o conjuntos de signos que nos serán útiles para realizar acciones determinadas. Algunas de las expresiones, como la de incremento (**++**) y el operador de disminución (**--**), nos permiten aumentar o disminuir en uno el valor almacenado de una variable. Sus equivalentes serían: **a += 1** / **b -= 1**. A continuación, veamos un ejemplo que nos demuestre la forma en que pueden escribirse los equivalentes:

```
c++;
c+=1;
c=c +1;
```

En este caso, las tres líneas de código realizan la misma acción: incrementar el valor en 1.

Operadores relacionales y de igualdad

Como vimos anteriormente, para hacer una comparación entre dos expresiones podemos utilizar operadores relacionales o de igualdad.

El resultado de una operación relacional es un valor booleano que solo puede ser verdadero o falso. Quizá necesitemos comparar dos expresiones para saber si son iguales o si una es mayor que otra.

RELACIONALES

EXPRESIÓN	ES EQUIVALENTE A
==	Igual a
! =	No es igual a
>	Mayor que
<	Menor que
> =	Mayor o igual a
<=	Menor o igual a

Tabla 5. Estos son los operadores relaciones y de igualdad que se pueden utilizar en C++.

Ejemplos de código:

```
(4==7)     / / se evalúa como falso.
(8>1)      / / se evalúa como verdadera.
(9!=2)     / / se evalúa como verdadera.
(7>=7)     / / se evalúa como verdadera.
(6<6)      / / se evalúa como falsa.
```

Desde luego, en vez de utilizar valores constantes, podemos recurrir a expresiones, variables u otro tipo de operaciones que deseemos comparar. Por ejemplo:

```
A=2;
B=3;
C=6;

(A==5)//se evalúa como falso ya que no es igual a 5.
(A*B>=C) // se evalúa como cierto, ya que (2 * 3> = 6) es verdadera.
(4+B>A*C) //se evalúa como falso desde (3 +4> 2 * 6) es falsa.
((B=2)==A)// se evalúa como verdadera.
```

Operadores lógicos

Al igual que vimos en otro lenguaje, el operador lógico nos devuelve dos posibles resultados: verdadero o falso.

Nosotros comenzaremos con uno de ellos, el operador **!** (signo de admiración), que nos devolverá el valor booleano opuesto a la evaluación de su operando. Por ejemplo:

ASIGNACIÓN

El operador = (signo igual) no es el mismo que el operador == (doble signo igual). Mientras que el primero es un operador de asignación (establece el valor en su derecha a la variable en su izquierda), el segundo es el operador de igualdad, que compara si ambas expresiones de sus extremos son iguales entre sí.

```
! (5==5)     // se evalúa como falso porque la expresión en su derecho
   (5==5) es cierto.
! (6<=4)     // se evalúa como verdadera, porque (6 <= 4) sería falsa.
! true  // se evalúa como falso
! false // se evalúa como verdadera.
```

Otros operadores lógicos que podemos utilizar en el lenguaje son **&&** (and) y **||** (or), que se usan en la evaluación de dos expresiones para obtener un resultado relacional único. El operador **&&** se corresponde con la operación lógica booleana **AND**, que da como resultado **true** si ambos operandos son verdaderos, y **false** en caso contrario.

LÓGICOS

A	B	A && B
true	true	true
true	false	false
false	true	false
false	false	false

Tabla 6. En este listado podemos ver una muestra de resultados para el operador **&&**, que evalúa la expresión **`A, B && : operador &&`**.

El operador **||** se corresponde con la operación lógica **OR** booleana, y da como resultado verdadero si cualquiera de sus dos operandos es cierto, y falso solo cuando ambos operandos también lo son.

MEMORIA

La memoria en las computadoras se organiza en bytes. Un byte es la cantidad mínima de memoria que podemos manejar en C++ y puede almacenar una cantidad relativamente pequeña de datos: un solo carácter o un entero pequeño (generalmente, un número entero entre 0 y 255).

LÓGICOS

A	B	A || B
true	true	true
true	false	true
false	true	true
false	false	false

Tabla 7. Dentro de los operadores lógicos, en este listado podemos ver los resultados posibles de **`A || B : operador ||`**.

Ambos operadores ya fueron utilizados por ejemplos de otro lenguaje, y es bueno tenerlos bien presentes, ya que nos servirán cuando empecemos a utilizar condiciones o estructuras repetitivas.

Operador condicional (?)

Este operador de C++ es aquel que evalúa una expresión y luego devuelve un valor. Si la expresión es considerada verdadera, devolverá un valor, y si es falsa, el valor será distinto. Su formato es:

Condición ? resultado1: resultado2

Si la condición es cierta, la expresión devolverá **resultado1**; en caso contrario, **resultado2**.

```
1 == 3? 2: 4     // devuelve 4, 1 no es igual a 3.
7 == 5 + 2? 4: 3  // devuelve 4, 7 es igual a 5+2.
4>2? a: b      // devuelve el valor de a, 4 es mayor que 2.
a>b?a:b      // devuelve el que sea mayor, a o b.
```

Por ejemplo, en C++ podemos escribir lo siguiente:

```
#include <iostream>
using namespace std;

int main ()
{
 int varA,varB,varC;

 varA=2;
 varB=7;
 varC = (varA>varB) ? varA : varB;

 cout << varC;

 getchar();
 return 0;

}
```

Operador coma (,)

Este operador se utiliza para separar dos o más expresiones que se incluyen en un mismo lugar. Cuando el conjunto de expresiones tiene que ser evaluado por un valor, solo se considerará la expresión ubicada más a la derecha. Por ejemplo, veamos el código siguiente:

```
varA = (varB=3, varB+2);
```

Primero se asigna el valor 3 a **varB**, y luego **varB+2** a la variable **varA**. Finalmente, la variable **varA** contendría el valor 5, y **varB**, el valor 3.

Operadores bitwise o bit a bit

Este operador se aplica en trabajos de lenguajes de bajo nivel, ya que trabaja a nivel de bit. Los operadores bit a bit modifican las variables teniendo en cuenta los patrones de bits que representan los valores almacenados. En la siguiente tabla veremos sus representaciones.

BITWISE

OPERADOR	EQUIVALENTE A	DESCRIPCIÓN
&	AND	Bitwise AND
|	OR	Bitwise Inclusive OR
^	XOR	Bitwise Exclusive OR
~	NOT	Unary complement (bit inversion)
<<	SHL	Shift Left – mover a la izquierda
>>	SHR	Shift Right – mover a la derecha

Tabla 8. Estos son los operadores **`bitwise`** de los que disponemos para activar o desactivar bits dentro de un entero.

Debemos considerar que una operación **bit a bit** o **bitwise** opera sobre números binarios a nivel de sus bits individuales. Es una acción primitiva rápida, soportada directamente por los procesadores.

Tipo de operador de conversión explícita

Así como en otros lenguajes contamos con funciones que nos permiten convertir algunas variables a otro tipo de datos, en el caso de C++ utilizaremos los paréntesis (**()**) que encierran una expresión, por ejemplo:

```
int varA;
float varF = 3,14;
varA = (int) varF;
```

Con este código de ejemplo convertimos el número **float** 3,14 a un valor **int** que sería 3. Los decimales se pierden, ya que el operador entre paréntesis (**int**) es entero y comprende números de esa índole. Otra forma de hacer lo mismo en C++ es utilizando la notación funcional

que precede a la expresión que se convierte por el tipo, y encierra entre paréntesis a dicha expresión, por ejemplo:

```
varA = int (varF);
```

Operador sizeof ()

Este operador ya fue utilizado en código de ejemplos anteriores; acepta un parámetro (que puede ser un tipo o una misma variable) y devuelve el tamaño en bytes de ese tipo u objeto:

```
varA = sizeof(char);
```

En el ejemplo, **Char** es un tipo largo de un byte, por lo que asignamos el **valor 1** a **varA**, y la información que devolverá **sizeof** es una constante que se determinará antes de la ejecución del programa. Recordemos que, anteriormente, lo aplicamos a un ejemplo para establecer el tamaño de los tipos de datos, por ejemplo:

```
cout << "El tamaño del int es:\t\t" << sizeof(int) << " bytes.\n";
cout << "El tamaño del short es:\t" << sizeof(short) << " bytes.\n";
```

Precedencia de los operadores

Si debemos codificar expresiones extensas, podemos dudar acerca de qué sector de nuestra sintaxis tiene que evaluarse primero. En la **Tabla 9** se hace un repaso de todos los operadores vistos.

PROCESADORES Y BITS

En procesadores simples de bajo costo, las operaciones de bit a bit, junto con las de adición y sustracción, son típica y sustancialmente más rápidas que la multiplicación y la división; mientras que en los procesadores de alto rendimiento las operaciones suelen realizarse a la misma velocidad.

OPERADORES

NIVEL	OPERADOR	DESCRIPCIÓN	AGRUPACIÓN
1	::	Bitwise AND	Left-to-right
2	() [] . -> ++ – dynamic_cast static_cast reinterpret_cast const_cast typeid	Alcance / ámbito	Left-to-right
3	++ – ~ ! sizeof new delete	Sufijo	Right-to-left
4	* &	Unary (prefijo)	
5	+ -	Indirecto y de referencia (punteros)	
6	(type)	Operador unary	
7	.* ->*	Convertir en tipo	Right-to-left
8	* / %	Puntero a member	Left-to-right
9	+ -	Multiplicación	Left-to-right
10	<< >>	Agregar	Left-to-right
11	< > <= >=	Cambiar	Left-to-right
12	== !=	Relacionar	Left-to-right
13	&	Igualdad	Left-to-right
14	^	bitwise AND	Left-to-right
15	|	bitwise XOR	Left-to-right
16	&&	bitwise OR	Left-to-right
17	||	logical AND	Left-to-right
18	= *= /= %= += -= >>= <<= &= ^= |=	logical OR	Right-to-left
19	,	Asignación	Left-to-right

Tabla 9. Aquí podemos observar los distintos tipos de operadores y el orden en el que C++ prioriza cada uno de ellos.

LA AGRUPACIÓN DEFINE EL ORDEN EN EL QUE LOS OPERADORES DEBEN EVALUARSE

La agrupación define el orden de precedencia en el que los operadores deben evaluarse, en caso de que, en una misma expresión, haya varios operadores del mismo nivel. Todos estos niveles de precedencia pueden manipularse o hacerse más legibles, eliminando posibles ambigüedades en el uso de paréntesis.

Hasta aquí hemos visto las declaraciones de variables, constantes y el uso de operadores, lo cual nos brindará las herramientas necesarias para empezar a desarrollar aplicaciones de distinta envergadura. A continuación, veremos las estructuras que podemos utilizar para darles sustento a nuestros algoritmos en C++.

Interactuar con el usuario

Hasta este punto, el código de ejemplo que hemos visto no realiza ninguna interacción con el usuario, salvo presionar la tecla **ENTER**. Es por eso que, para enriquecer nuestras aplicaciones, es importante conocer cuáles son los métodos de entrada y salida básicos que ofrecen las librerías de C++.

Estas librerías están incluidas en el archivo **iostream**, que declaramos en el encabezado de los desarrollos.

Cout - salida estándar

La palabra reservada **cout** es utilizada por C++ para definir que deseamos dar salida de información por pantalla. Se utiliza conjuntamente con el operador de inserción, que se escribe como **<<** (dos signos "menor que").

```
#include <iostream>
using namespace std;

int main ()
```

```
{
 int varA=0;
 cout << "La sentencia de salida es: " ; //muestra La sentencia de salida, en
    la pantalla
 cout << 120; // muestra el número 120 en la pantalla
 cout << varA; // muestra el contenido de varA en la pantalla;
 getchar();
 return 0;
}
```

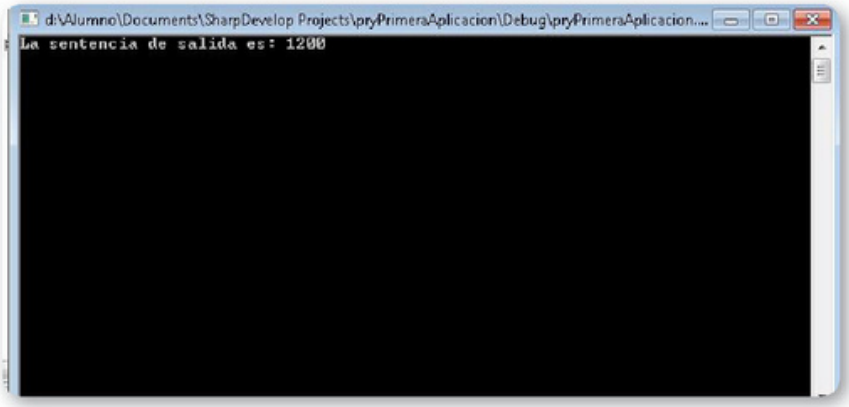


Figura 9. Resultado del ejemplo anterior, donde vemos que podemos mostrar información en pantalla desde una variable.

Como podemos ver y probar en código, la sintaxis **<<** es la que nos da la posibilidad de introducir datos. Es importante tener cuidado cuando utilizamos cadenas de caracteres, es decir, un texto que deseemos mostrar en pantalla, porque debemos encerrar siempre ese texto entre comillas **“”**. Por ejemplo:

```
cout << "Prueba";
cout << Prueba;
```

De esta forma, en la primera sentencia veremos que la pantalla muestra la palabra **Prueba**; y en la segunda se reconocerá **Prueba** como variable y se mostrará su contenido.

Teniendo en cuenta que para los textos debemos aplicar comillas dobles, ahora veremos que podemos utilizar el operador de inserción **<<** más de una vez en una sola sentencia, por ejemplo:

```
cout << «Estoy» << «juntando» << «varias cadenas»;
```

Frente a este proceso que vimos anteriormente como **concatenar** (unir varias palabras en una sola), notaremos que ahora el procedimiento no nos será útil si deseamos utilizar variables y textos. Veamos el ejemplo:

```
int varAltura=180;
int varPeso=90;
cout << "La altura es: " << varAltura << ". El peso es: " << varPeso;
```

El resultado sería: La altura es 180. El peso es: 90.

Debemos tener en cuenta que **cout** no agrega un salto de línea después de su salida, a menos que lo indiquemos expresamente. De esta forma, los ejemplos serían:

```
cout << "Esta es una sentencia." ;
cout << "Esta es otra frase." ;
```

Si deseamos realizar el salto de línea por medio de código, debemos saber que en C++ un carácter de nueva línea se puede especificar con \n (barra invertida, n). En este caso, el código será el siguiente:

```
cout << "Primera sentencia \n." ;
cout << "Segunda sentencia \n Tercera sentencia.";
```

El resultado será una oración por renglón. Para ver mejor este ejemplo, debemos codificarlo en C++.

En este sistema también podemos encontrar la palabra reservada **endl**, que permite realizar el salto de línea. Si tomamos el ejemplo anterior, el código nos quedaría:

```
cout << "Primera sentencia" << endl;
cout << "Segunda sentencia " << endl;
cout << "Tercera sentencia " << endl;
```

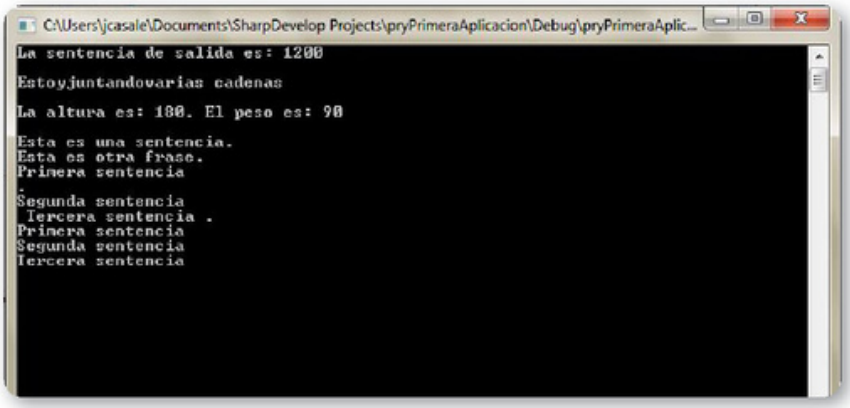


Figura 10. Resultado de todos los ejemplos; podemos diferenciar el manejo de cadenas y concatenación.

Hemos repasado las opciones que tenemos para mostrar información en pantalla desde C++; ahora continuaremos con el ingreso de datos por medio de una aplicación.

Cin – entrada estándar

El ingreso de datos estándar se da por medio del teclado, y la norma para realizar su expresión es la palabra reservada **cin** y los símbolos **>>**. A continuación, veamos un ejemplo:

```
int edad;
cin >> edad;
```

La función de estas sentencias es solucitarle al usuario el ingreso de un dato, en este caso la edad. Una vez que el dato haya sido ingresado, el sistema lo cargará directamente en la variable. Es importante tener en cuenta que **cin** tomará los valores ingresados por teclado, siempre que el usuario haya presionado la tecla **ENTER**.

Recordando los tipos de variables, si utilizamos un tipo entero con **cin**, solicitando en este caso un número, lo que debemos ingresar es un valor numérico. Si ingresamos otro tipo de valor, es probable que nos aparezca algún error de conversión o desbordamiento de variable.

Es por eso que debemos prestar mucha atención en el tipo de variable que vayamos a utilizar para la aplicación. Veamos el siguiente ejemplo:

```
int i;
cout << "Por favor, introduzca un valor entero:" ;
cin >> i;
cout << "El valor que ha introducido es" << i;
cout << "y su doble es" << i * 2 << "\ n". ;
```

También podemos utilizar **cin** para solicitarle al usuario dos datos, escribiendo esta sentencia en una única línea, por ejemplo:

```
cin >> varA >>    varB;
```

El programa le solicitará al usuario un valor para la primera variable y, luego, otro para la siguiente.

Operaciones con cadenas de texto

Para obtener por pantalla lo que el usuario ingresa, podemos utilizar la función **cin**. En este caso, es importante tener en cuenta que esta solo reconocerá lo escrito hasta que aparezca un espacio. Debido a las complicaciones que esto trae en las cadenas extensas de texto, se recomienda utilizar la función **getline()**, que permite capturar mayor cantidad de texto. Veamos un ejemplo:

OPERADORES UNARY

Este operador solo tiene un operando; esto quiere decir que puede contener un objeto, variable u operando específico del lenguaje. Por ejemplo: **Sizeof() / New() / Delete() / (type)/ ++ / -- / ~ / ! / + / -** .
Ejemplo: **Sizeof(short) / (int) variable / variable++ / --variable.**

```
#include <iostream>
#include <string>
using namespace std;
int main ()
{

  string mystring;
  cout << "Cuál es tu nombre? ";
  getline (cin, mystring);
  cout << "Hola " << mystring << ".\n";
  cout << "Cuál es tu equipo favorito de futbol? ";
  getline (cin, mystring);
  cout << "Yo soy fanatico de " << mystring << " también!\n";
  getchar();
  return 0;
}
```

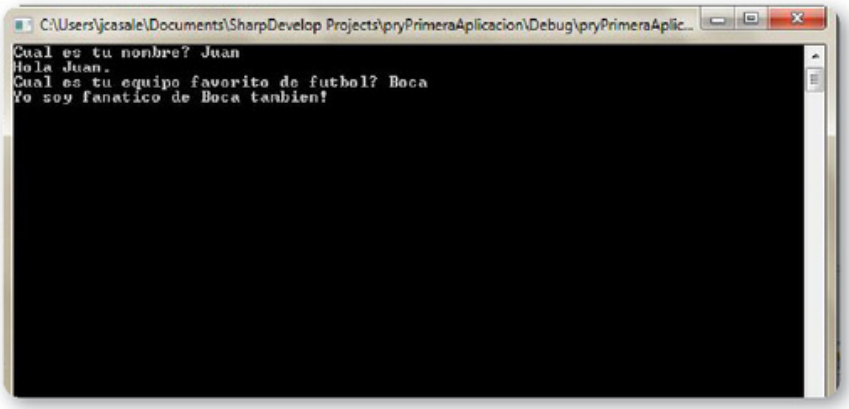


Figura 11. Si ingresamos en la pantalla los datos requeridos, el resultado del ejemplo es el que vemos en la imagen.

Podemos ver que **getline** requiere dos parámetros: uno es la instrucción **cin** y otro es la **variable** donde guardamos la información.

Ahora que ya hemos visto el funcionamiento de todas estas sentencias de entrada y salida de información estándar en C++, podemos avanzar en las instrucciones que nos serán útiles para realizar algoritmos más complejos.

Todo tiene un orden en la programación

Ya sabemos cómo se pueden utilizar los datos en C++, ahora vamos a ver cómo se pueden realizar diferentes acciones o tomar distintos rumbos en un algoritmo de programación. Para hacerlo, vamos a usar sentencias parecidas a las que estudiamos anteriormente en las **estructuras de control**, y conoceremos las estructuras secuenciales y repetitivas. Es importante considerar que en C++ no siempre se trabajará de manera secuencial, por lo que debemos diferenciar los bloques de código con llaves { }. A continuación, veamos cómo se aplica este concepto en las siguientes estructuras.

Estructura condicional

La estructura condicional **if** se usa para ejecutar una condición si la expresión se cumple. Por ejemplo:

```
if (varX == 100)
  cout << "varX es 100";
```

La estructura es muy parecida a otros condicionales que vimos anteriormente, y su sintaxis es sencilla:

```
If (condición)
      Instrucciones...
```

Si necesitamos realizar más de una instrucción para que se ejecute, debemos especificar un bloque con las llaves **{ }**, por ejemplo:

```
if (varX == 100)
{
  cout << "varX es ";
  cout << varX;
}
```

Para conocer la veracidad o no de una condición y tomar medidas si esta es verdadera o falsa, utilizamos la siguiente sintaxis:

```
if (condición)
        Instrucciones...
else
        Instrucciones...
```

Veamos un ejemplo:

```
if (varX == 100)
  cout << "varX es 100";
else
  cout << "varX no es 100";
```

Para concatenar o anidar las estructuras **if**, primero debemos ver cómo es la estructura:

```
if (condición)
        Instrucciones...
elseif
        Instrucciones...
else
        Instrucciones...
```

```
#include <iostream>
#include <string>
using namespace std;

int main ()
{
int varX=0;
varX=7;

if (varX > 0)
  cout << "varX es positivo";
```

```
else if (varX < 0)
  cout << "varX es negativo";
else
  cout << "varX es 0";

getchar();

}
```

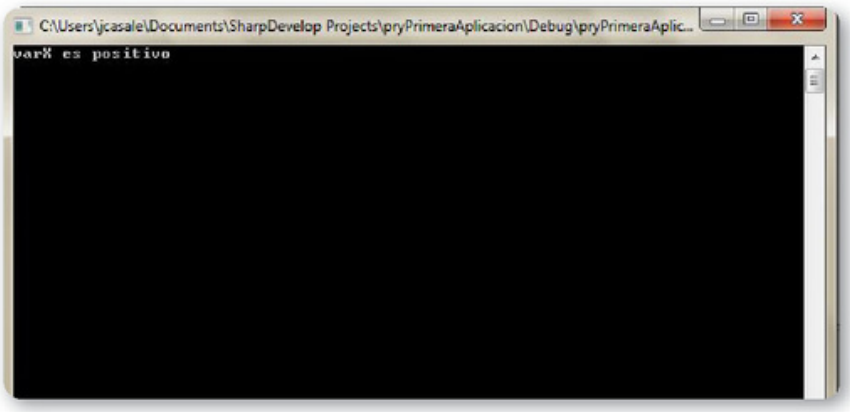


Figura 12. Resultado del ejemplo si el valor de **varX** es igual a 7 y luego es comparado en la estructura **if**.

Si vamos a realizar más sentencias de instrucciones, es importante recordar que tenemos que encerrarlas entre llaves **{}**.

Estructuras selectivas (switch)

Esta estructura selectiva se usa para valorar una condición y, dependiendo de su resultado, se pueden dar distintos casos y resoluciones. Su sintaxis es la siguiente:

```
switch (condición)
{
  case constante:
     instrucciones...;
```

```
    break;
  case constante:
    instrucciones...;
    break;
  .
  .
  .
  default:
    default instrucciones...
}
```

En este caso, notaremos que la estructura utiliza la palabra reservada **break**, que indica el fin de las instrucciones del **case**. A continuación, veamos un ejemplo para comparar ambas.

Switch	If-else

switch (x) {
case 1:
cout << "x es 1";
break;
case 2:
cout << "x es 2";
break;
default:
cout << "El valor de x es desconocido";
}
	if (x == 1) {
	cout << "x es 1";
	}
	else if (x == 2) {
	cout << "x es 2";
	}
	else {a
	cout << " El valor de x es
	desconocido";
	}

Tabla 10. Comparación de sintaxis y funcionalidad de estructuras condicionales en el lenguaje.

SWITCH EN C# Y C++

En C# la instrucción **switch** cumple el mismo propósito que en C++. Sin embargo, en C# es más potente, ya que, a diferencia de C++, nos permite utilizar una cadena como variable de selección. En la sintaxis de ambos lenguajes debemos indicar una salida explícita del control de flujo del **switch**.

Estructuras repetitivas (loop)

Recordemos que estas estructuras son las que utilizamos para repetir acciones hasta lograr que se cumpla una determinada condición. A continuación, veamos cuáles son:

While / Loop

La sintaxis es:

while (expresión) instrucciones

Su función es repetir la declaración, siempre y cuando la condición establecida en la expresión sea verdadera. Por ejemplo, vamos a hacer un programa de la cuenta regresiva mediante un bucle **while**:

```
#include <iostream>
using namespace std;

int main ()
{
  int n;
  cout << "Ingrese un número de inicio > ";
  cin >> n;

  while (n>0)
  {
    cout << n << ", ";
    --n;
  }

  cout << "SE TERMINO!\n";
  return 0;
}
```

Al momento de iniciar el programa, se le pide al usuario que introduzca un número de inicio para la cuenta regresiva, y así se da comienzo al tiempo de bucle. Si aquel valor especificado por el usuario cumple la condición **n > 0** (donde **n** es mayor que cero), el bloque que

sigue a la condición se ejecutará y se repetirá, siempre y cuando la condición (**n > 0**) continúe siendo verdadera.

Cuando se crea un bucle **while**, este debe terminar en algún momento. Para eso, tenemos que escribir alguna instrucción o método que determine la condición como falsa; de lo contrario, el bucle se convertirá en lo que se conoce como "**bucle infinito**", es decir, girará dentro de él para siempre. En este caso hemos incluido **–n** para disminuir el valor de la variable que está siendo evaluada en la **condición**. De esta forma, eventualmente haremos que la condición (**n > 0**) llegue a ser falsa después de un cierto número de iteraciones en el bucle.

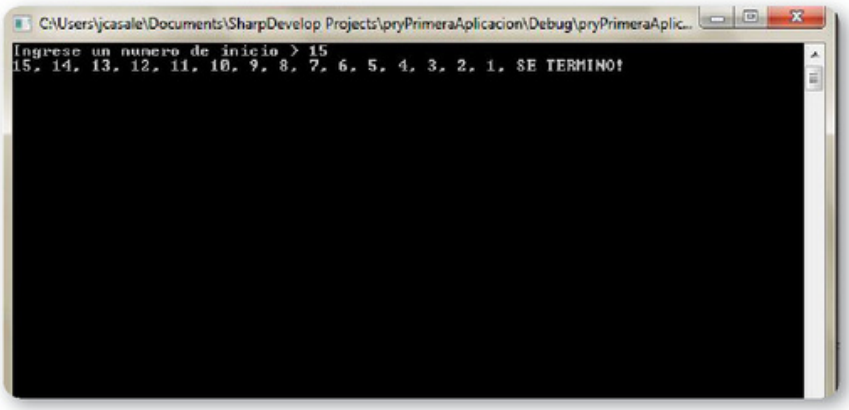


Figura 13. Resultado del ejemplo donde se realiza una cuenta atrás gracias a la estructura repetitiva.

For / Loop

La sintaxis de esta estructura repetitiva es:

```
for(inicialización; condición; incremento) instrucciones;
```

Su principal función es repetir las instrucciones mientras que la **condición** se cumpla (como lo hacía el **while/loop**), y además, proporcionar lugares específicos para contener una **inicialización** y un **incremento**. Este bucle está especialmente diseñado para realizar una acción repetitiva, utilizando un contador que se declara e inicializa en la estructura e incrementa su valor en cada iteración. Repasemos qué significa cada parte de esta sintaxis:

- **Inicialización**: ajuste de valor inicial para una variable de contador, que se ejecuta solo una vez.
- **Condición**: se chequea a medida que el bucle se produce: si es verdadera, el bucle continúa; si es falsa, se termina.
- **Instrucciones**: se ejecutan en cada entrada al bucle; puede ser una sola sentencia o un bloque entre llaves {}.
- **Incremento**: cada vez que chequea la condición, se lleva a cabo lo que especifiquemos en el incremento o decremento de una variable.

A continuación, veamos cómo sería en código, haciendo referencia a la condición del ejemplo anterior:

```
#include <iostream>
using namespace std;
int main ()
{
  for (int n=10; n>0; n--)
  {
    cout << n << ", ";
  }
  cout << "CUENTA FINALIZADA!\n";
  return 0;
}
```

Tengamos en cuenta que tanto la inicialización como el incremento son opcionales; por lo tanto, no es necesario declararlos para el uso de la estructura **For**. Dicha estructura también nos permite utilizar más de una expresión dentro de los paréntesis que siguen al **For**, gracias al uso de **comas**. A continuación, veamos un ejemplo en el cual podemos inicializar más de una variable:

```
for (n=0, i=100 ; n!=i ; n++, i--)
  {
    cout << n << ", ";
  }
```

Hay que tener cuidado al utilizar las comas, y punto y coma, ya que estos últimos dividen la expresión.

```
for (n=0, i=100 ; n!=i ; n++, i--)
{
 Instrucciones...
}
```

En la figura anterior vimos cómo estaría dividida la expresión con comas, y punto y coma. Esta condición pregunta si **n** es distinto de **i**; en el caso de ser iguales, el bucle se detendrá.

Ejercitación de entrada/salida de datos y uso de for

En el caso práctico que aparece a continuación, pondremos a prueba lo que venimos aprendiendo. Para esto, vamos a usar el operador **XOR** y la función **getline()** para encriptar un texto que haya ingresado el usuario (incluyendo los espacios).

1. Creamos un proyecto en **SharpDevelop**, con el nombre **EjemploFOR**.

2. Codificamos lo siguiente:

```
#include <iostream>
#include <string>

using namespace std;

int main()
{
 int longitud;
 const int NUMERO=120; //número máximo de caracteres.
 char texto[NUMERO],key;

 cout << "Ingrese una oración:";
 cin.getline(texto,120); //toma el renglón del texto.
 cout << "Ingrese la clave para encriptar (un digito):";
 cin >> key;
```

```
longitud=strlen(texto);
cout << "\n\t*** Texto Encriptado ***\n";

for (int i=0;i<longitud;i++)
{
        texto[i] = texto[i] ^ key; //XOR
}

cout << texto << endl;
cout << "\n\t*** Texto DeCodificado ***\n";

for (int i=0;i<longitud;i++)
{
        texto[i] = texto[i] ^ key;
}

cout << texto << endl;

system("pause");
}
```

Antes de continuar con otras estructuras, revisemos lo que utilizamos en este código de ejemplo:

a. Utilizamos variables y constantes. Recordemos que podemos abreviar las variables en el caso de los ejemplos que vimos. En general, contienen un nombre completo, pero no es necesario que sea extenso; eso dependerá del criterio de cada desarrollador.

PRINTF

Para la salida de información en pantalla, podemos utilizar la función **printf**, usando una cadena dentro de esta función de la siguiente forma:

printf("Primera cadena" "Segunda cadena");

printf("Primera cadena" texto en medio "Segunda cadena");

printf("Esto es \"extraño\"");

b. Aplicamos sentencias de entrada y salida de datos. Dentro de estas últimas, contamos con una librería llamada **string**, nos permite utilizar la función **strlen()**.

c. Utilizamos una función cíclica o bucle: el **for**. Nos permite recorrer todo el texto y encriptarlo.

d. Aplicamos una nueva palabra reservada system("pause"). Nos muestra un texto en la consola que dice "presione cualquier tecla para continuar" y, así, nos permite ver el resultado de este ejemplo.

De esta forma, el ejemplo quedaría como se observa en la **Figura 14**.

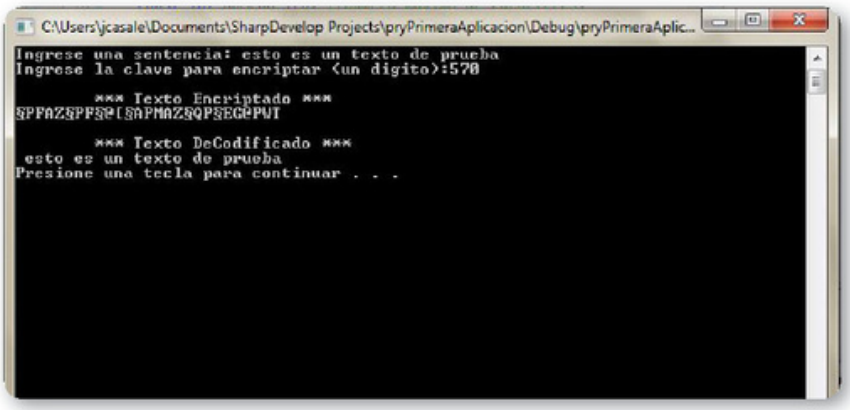


Figura 14. Resultado del ejercicio, donde podemos ver la forma en que se encripta un texto con C++.

Salto de declaraciones

Los saltos de declaraciones son útiles al momento de realizar algunas instrucciones que consideramos importantes, y que no necesitemos que continúe preguntándonos por una condición en un bucle.

Sentencia break

Utilizando la palabra **break** podemos terminar un bucle cuando una variable tome un determinado valor aun si la condición no ha finalizado. Es decir, podemos utilizarlo para poner fin a un bucle infinito, o para forzarlo a terminar antes de su fin natural.

En el siguiente ejemplo, vamos a ver cómo se detiene una cuenta regresiva antes de que llegue su fin natural:

```
#include <iostream>
using namespace std;

int main ()
{
  int n;
  for (n=10; n>0; n--)
  {
    cout << n << ", ";
    if (n==3)
    {
      cout << "cuenta regresiva abortada!";
      break;
    }
  }
  getchar();
  return 0;
}
```

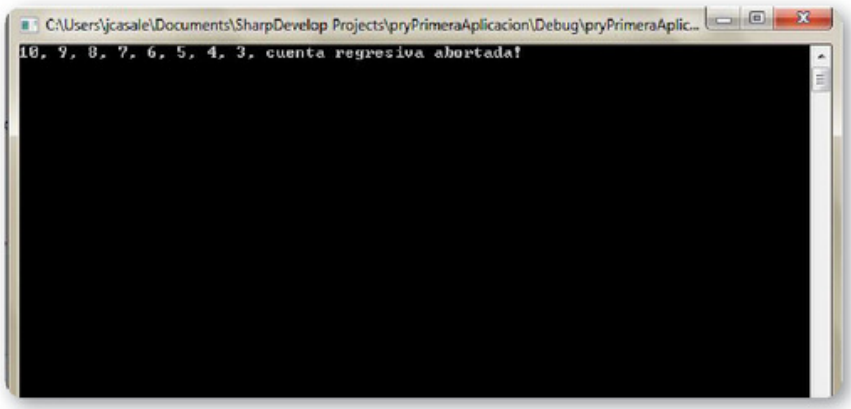


Figura 15. Resultado del ejercicio cortando o deteniendo un proceso repetitivo con una estructura condicional.

Es bueno tener en cuenta que podemos utilizar un **break** en cualquier punto del código dentro de un bucle (o repetitiva). Recordemos que saldrá del bloque de código donde es utilizado, y continuará con las sentencias que le siguen a la estructura repetitiva.

Sentencia continue

La instrucción **continue** realiza un "salto" en el bloque de código que nos llevará al extremo de las instrucciones. Se utiliza en estructuras repetitivas para "saltar" algunos procesos que especifiquemos y, así, continuar con las iteraciones. En el ejemplo que aparece a continuación, vamos a saltar el número 2 en la cuenta regresiva:

```
#include <iostream>
using namespace std;

int main ()
{
  for (int n=10; n>0; n--) {
    if (n==2) continue;
    cout << n << ", ";
  }
  cout << "FINALIZADO!\n";
  return 0;
}
```

Si pusiéramos a prueba este código, el resultado sería el siguiente: 10, 9, 8, 7, 6, 5, 4, 3, 1, FINALIZADO!

Sentencia goto

Esta sentencia podemos utilizarla para realizar un salto a un sector del algoritmo que estemos desarrollando. Antes de ejecutarla, es importante ser cuidadosos con ella, porque puede hacer el salto y ejecutar tipos incondicionales.

CONTROL DE FLUJO EN C# Y C++

En el control de flujo de la programación de estos lenguajes hay algunas diferencias sintácticas en las **instrucciones if**, **while**, **do/while** y **switch**. A su vez, existen otras compartidas por ambos lenguajes, como son: **for**, **return**, **goto**, **break** y **continue**. En C# encontraremos la adicional **foreach**.

El punto de destino siempre será rotulado por un texto seguido de dos puntos (:). En el siguiente ejemplo, vamos a ver un bucle "cuenta atrás" utilizando la sentencia **goto**:

```
#include <iostream>
using namespace std;

int main ()
{
  int n=10;
  loop:
  cout << n << ", ";
  n--;

  if (n>0) goto loop;
  cout << "FINALIZO!\n";
  return 0;
}
```

Para tener en cuenta, esta instrucción no posee ningún uso concreto en la programación orientada a objetos o estructurada.

Función Exit

Esta función puede encontrarse en otros lenguajes y tiene como objetivo finalizar el programa. Está en la librería **cstdlib** de C++, y su estructura es la siguiente:

void exit (int exitcode);

UTILIZAR GOTO

Las sentencias **goto** no deben usarse cuando se resuelven problemas mediante programación estructurada. Existen mecanismos suficientes para hacer de otro modo todo aquello que pueda efectuarse mediante **goto**. La metodología de programación actual nos lleva a utilizar otras herramientas de saltos.

Datos estructurados: arrays

En esta sección aprenderemos a manejar arrays, estructuras que nos será muy útiles para utilizar y almacenar información en memoria, durante un período de tiempo necesario en el desarrollo de nuestros programas. Este tipo de estructuras permiten definir vectores, matrices, tablas y estructuras multidimensionales.

Manejo de arrays

Un **array** es una serie de elementos homogéneos que se grabarán en espacios continuos de memoria, a los cuales podemos acceder en forma individual gracias a un índice. Su modo de representación se observa en la **Figura 16**.

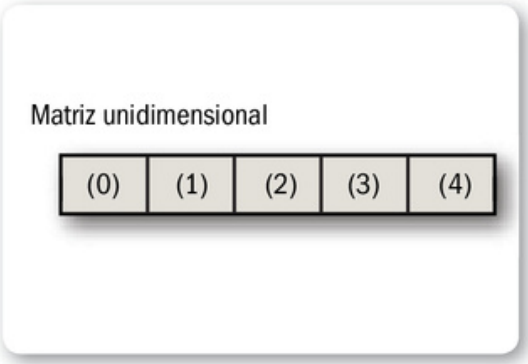


Figura 16. En este ejemplo podemos ver la forma correcta de representar un array de cinco posiciones.

Cada ubicación en blanco representa un elemento en el array y, si no le definimos ningún valor, este tomará por defecto el que indica el lenguaje. Recordemos que todos los elementos serán del mismo tipo de dato (de allí su característica de homogéneo).

El índice en C++ inicia en 0 a n; por eso, en el array que vimos en la imagen es 0 a 4, porque se obtienen 5 espacios para grabar datos.

A continuación, veamos cómo se declaran los arrays:

```
tipo nombre [elementos];
```

Como podemos notar, la declaración es como una variable que usualmente utilizamos en C++. Veamos cómo se conforma su sintaxis:

- **tipo**: se refiere a un tipo válido (**int** , **float**, **char**...)
- **nombre**: es un identificador válido.
- **elementos**: son espacios que se declaran en el array y siempre aparecen entre corchetes **[ ]**.

Veamos una declaración en C++:

```
int miArray [5];
```

El elemento de campo encerrado entre corchetes **[ ]** representa el número de elementos de la matriz y debe ser un valor constante, ya que las matrices son bloques de memoria no dinámica, cuyo tamaño tiene que ser determinado antes de la ejecución.

Sintaxis de inicialización

Dentro de las funciones, como puede ser el **main()**, podemos declarar arrays locales. Estos no tendrán un valor definido si no se inicializan por código, por lo tanto, su contenido será indefinido hasta que le almacenemos un valor.

En el caso de las matrices globales y estáticas, los elementos se inicializan automáticamente con valores por defecto; esto significa que estarán llenos de ceros. En ambos casos, local y global, cuando se declara una matriz, podemos asignar valores iniciales a cada uno de sus elementos por los valores que encierra entre las llaves **{ }**.

Veamos un ejemplo:

BOO

Boo es un lenguaje de programación orientado a objetos de tipo estáticos para la Common Language Infrastructure, con una sintaxis inspirada en Python y el énfasis puesto en la extensibilidad del lenguaje y su compilador. Se integra sin fisuras con Microsoft.NET y Mono y destacamos que es de código libre. Podemos visitar su página oficial en http://boo.codehaus.org.

```
int miArray[5] = {1999, 64, 180, 201, 5138};
```

Para no provocar desbordamientos en el array, no debemos agregar más elementos entre **{}** de los que hay declarados en **[ ]**.

Es útil tener en cuenta que podemos declarar arrays sin especificar su tamaño y asignar valores, que a su vez tomarán el tamaño de la cantidad de elementos agregados, por ejemplo:

```
int miArray[] = {1999, 64, 180, 201, 5138};
```

Como podemos ver, en esta declaración quedaría entonces el array en un tamaño de 5 espacios.

Acceder a valores

El acceso a estas estructuras es muy similar a lo que vimos en otro lenguaje: debemos especificar el índice donde está almacenado el valor.

Podemos acceder al array de forma individual y, entonces, leer o modificar los valores que queramos. La sintaxis es sencilla:

nombre [indice]

Por lo tanto, si deseamos grabar el valor 2012 en el inicio del array, la declaración sería la siguiente:

```
miArray[0] = 2012;
```

RUBY

Ruby es un lenguaje de programación dinámico y de código abierto enfocado en la simplicidad y la productividad. Su creador, Yukihiro Matsumoto, mezcló partes de sus lenguajes favoritos para formar un nuevo lenguaje que incorporara tanto la programación funcional como la programación imperativa. Podemos visitar la página oficial en **www.ruby-lang.org/es**.

Y si precisamos obtener un valor del array, debemos asignarlo a una variable que sea del tipo de dato correspondiente a la estructura.

```
varA = miArray[0];
```

Como podemos ver, los corchetes **[ ]** representan dos tareas diferentes que no debemos confundir: por un lado, especifican el tamaño de las matrices que hayan sido declaradas; y por el otro, definen los índices de elementos de la matriz de hormigón.

```
int miArray[5];  // declaración de un array
miArray[2] = 180; // acceso a un dato/espacio del array
```

A continuación veremos cuáles son las sintaxis válidas para trabajar con arrays en C++:

```
miArray[0] = varA;
miArray[varA] = 75;
varB = miArray[varA+2];
miArray[miArray[varA]] = miArray[2] + 5;
```

Arrays multidimensionales

Los arrays de arrays, o también conocidos como multidimensionales, son estructuras bidimensionales muy parecidas a las hojas de cálculo. Veamos en la **Figura 17** cómo podría ser su apariencia.

C Y EL DESARROLLO

Este lenguaje ha sido estrechamente ligado al sistema operativo UNIX, puesto que fueron desarrollados conjuntamente. Sin embargo, no está ligado a ningún sistema operativo ni máquina en particular. Se lo suele llamar lenguaje de programación de sistemas debido a su utilidad para escribir compiladores y sistemas operativos, aunque de igual modo se puede desarrollar cualquier tipo de aplicación.

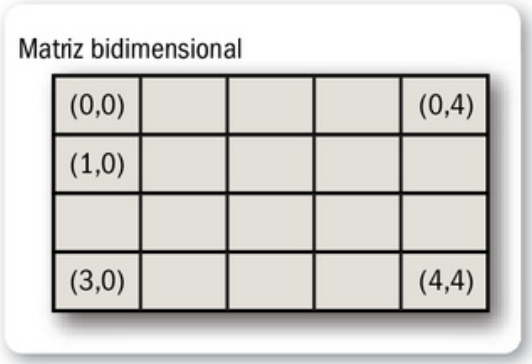


Figura 17. En este ejemplo podemos ver la forma correcta de representar un array de 2 dimensiones.

El array que vemos en la imagen representa una estructura de 4 por 5 (4 filas y 5 columnas). Para declararlo, deberíamos escribir:

```
int miArray[4][5];
```

Si queremos hacer referencia a un espacio dentro de este array, debemos codificar lo siguiente:

```
miArray[2][1];
```

Si bien los arrays multidimensionales pueden tener más de un índice, debemos tener cuidado, ya que si tienen más dimensiones y espacios, mayor será la cantidad de memoria requerida. Por ejemplo:

LIBRERÍAS

Una librería es un conjunto de recursos (algoritmos) prefabricados, que pueden ser utilizados por el desarrollador para realizar determinadas operaciones. Las declaraciones de las funciones usadas en ellas, junto con algunas macros y constantes predefinidas que facilitan su manejo, se agrupan en ficheros de nombres conocidos que suelen encontrarse en sitios predefinidos.

```
int miArray[100][50][25][10];
```

Recorrer arrays

Es importante pasar por cada espacio de dato que hayamos declarado; al ser esta una tarea repetitiva, podemos utilizar cualquiera de las estructuras de bucle que ya conocemos. En el ejemplo que aparece a continuación, cargaremos datos en un array de 3x5:

```
#include <iostream>
using namespace std;

#define COLUMNA 5
#define FILA 3

int miArray [FILA][COLUMNA];
int n,m;

int main ()
{
 for (n=0;n<COLUMNA;n++)
   for (m=0;m<FILA;m++)
   {
     miArray[n][m]=(n+1)*(m+1);
cout<<"Fila: "<<n << " / ";
cout<<"Columna: "<<m << " / ";
cout<<"Dato: "<<miArray[n][m];
```

C

Es un lenguaje de programación de propósito general que ofrece economía sintáctica, control de flujo, estructuras sencillas y un buen conjunto de operadores. Al no estar especializado en ningún tipo de aplicación, se convierte en un lenguaje potente sin límites en su campo de aplicación. Fue creado en la misma época de UNIX.

```
        cout<<endl;
          }
        getchar();
        return 0;
      }
```

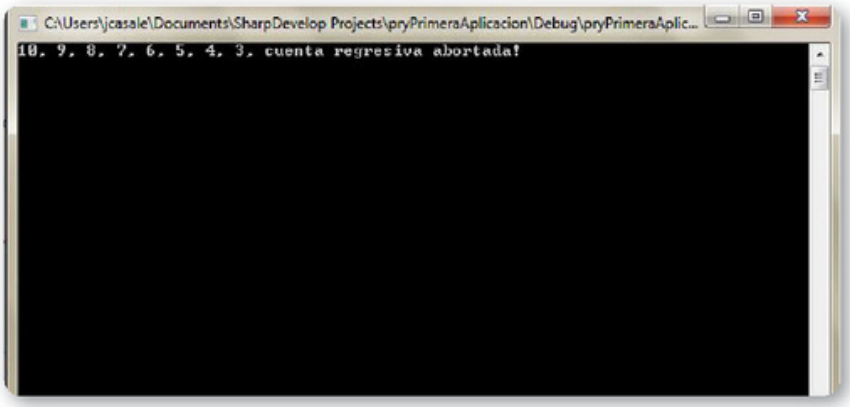


Figura 18. Recorrido y carga de un array con estructuras repetitivas, mostrando los resultados en pantalla.

RESUMEN

En el transcurso de este capítulo, nos encontramos con un lenguaje más "duro", pero que, en esencia de programación, es similar a otros. Iniciamos utilizando un IDE gratuito, donde se pueden crear proyectos de consola y practicar la declaración de variables, junto con el uso de operadores en C++. Luego aprendimos a capturar información desde la consola y a realizar todo el proceso de información necesario para devolverle al usuario los mensajes correspondientes. Finalmente, hicimos uso de las estructuras array que nos permiten almacenar y manipular gran cantidad de datos en memoria.

Después de todo, es bueno tener en cuenta que la base del desarrollo siempre estará en nuestra lógica, que luego será bajada al lenguaje de programación.

Actividades

TEST DE AUTOEVALUACIÓN

1. ¿Qué ventajas se obtienen al utilizar C++ sobre otros lenguajes?
2. ¿Qué entorno gráfico utiliza el lenguaje C++?
3. ¿Qué es SharpDevelop y Mono Project?
4. ¿Cómo se declaran variables y constantes en C++?
5. ¿Qué significa el "encabezado" en el archivo que se crea de C++?
6. ¿Qué significan **#include** y **using**?
7. Describa lo que es un literal en las declaraciones de variables.
8. Nombre los tipos de datos más frecuentes en programas de C++.
9. ¿Qué es un operador Bitwise?
10. ¿Cómo es posible convertir tipos de datos en C++?

ACTIVIDADES PRÁCTICAS

1. Realice un programa de consola con SharpDevelop que le permita al usuario acceder a datos, para realizar sumas y restas, mostrando el resultado mientras se generan los cálculos.
2. Efectúe una aplicación que genere una serie de números aleatorios entre 0 y 100 por cada letra que ingrese el usuario en la consola. Use la función **rand()**, que genera un número entero aleatorio entre 0 y 32767. La función se encuentra en math.h, y debe incluirla en el programa.
3. Con la función **atoi(char *)**, que se encuentra en la librería **stdlib.h**, convierta sobre el mismo proyecto una cadena de caracteres en un número entero. No olvide que la directiva **#include** incluirá tanto las librerías **math.h** como **stdlib.h**.
4. Realice una aplicación para la simulación de dados de 6 caras, donde deberá mostrarse el valor del dado cada vez que el usuario ingrese la palabra "tirar".
5. Realice una aplicación estilo "el ahorcado" donde se muestren los espacios de una palabra (a su elección). Si el usuario ingresa mal una letra, se mostrará su error y el programa le pedirá que lo vuelva a escribir. Si ingresa "ver" debe mostrarse el historial de intentos, y con "fin", terminar el programa.

*06

Estructuras de datos en la programación

En este capítulo comenzamos a profundizar en la algoritmia de la programación de mano del potente lenguaje C++. Al analizar el funcionamiento interno de los datos, vamos a conocer el lado más rígido de este proceso. De esta forma, nos dirigimos hacia un nivel avanzado que constituye la base fundamental del desarrollo.

- **Tipos de estructuras 252**
  - Datos simples y estructurados 253
  - Estructuras dinámicas y estáticas .. 256
  - Estructuras dinámicas y punteros... 257
- **Listas 268**
  - Listas enlazadas 269
  - Listas doblemente enlazadas 293
- **Pila 312**
  - Crear una pila 313
  - Eliminar en una pila (pop) 318
  - Buscar elementos en una pila 320
- **Cola 323**
  - Crear una cola 325
  - Eliminar elementos de una cola 328
  - Buscar elementos en una cola 332
- **Resumen 333**
- **Actividades 334**

Tipos de estructuras

En capítulos anteriores hemos utilizado distintos tipos de estructuras de datos, pero no conocíamos su categorización en profundidad. En la imagen de la **Figura 1**, veremos cómo se dividen estas estructuras.

Datos simples		
Estándar	Entero (integer)	Homogéneas
	Real (real)	Homogéneas
	Caracter (char)	Homogéneas
	Lógico (boolean)	Homogéneas
Definido por el usuario	Subrango (subrange)	Homogéneas
	Enumerativo (enumerated)	Homogéneas

Datos estructurados		
Estáticos	Arreglo Vector	Homogéneas
	Arreglo Matriz	Homogéneas
	Registro	Heterogéneas
	Conjunto	Heterogéneas
	Cadena	Heterogéneas
	Cadena	Heterogéneas
Dinámicos	Lista (pila / cola)	Heterogéneas
	Lista enlazada	Heterogéneas
	Árbol	Heterogéneas
	Grafo	Heterogéneas

Figura 1. Estas son las estructuras de datos más frecuentes utilizadas en los diferentes lenguajes de programación.

Hemos utilizado los datos **simples** en los capítulos en que tratamos el pseudocódigo, Visual Basic y C++, practicando continuamente la declaración de variables. En este capítulo utilizaremos los datos

que nos falta estudiar, los **estructurados**. A la hora de definir su concepto, podemos decir que la estructura de datos es una colección (normalmente de tipo simple) que se distingue por tener ciertas relaciones entre los datos que la constituyen.

Datos simples y estructurados

Los datos de tipo **simple** tienen una representación conocida en términos de espacio de memoria. Sin embargo, cuando nos referimos a datos **estructurados**, esta correspondencia puede no ser tan directa. Por eso, vamos a hacer una primera clasificación de los datos estructurados en: **contiguos** y **enlazados**.

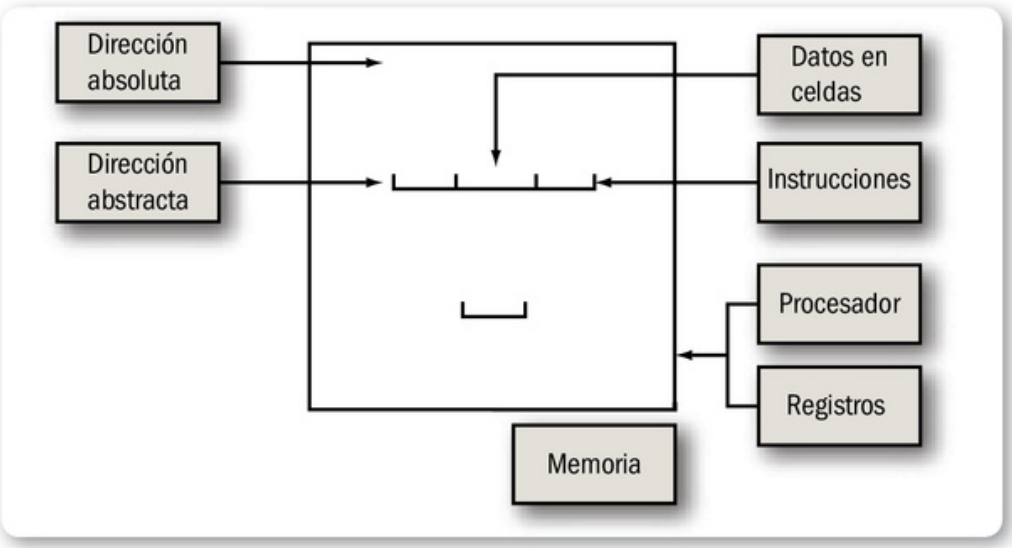


Figura 2. En esta imagen vemos el proceso involucrado en el almacenamiento de una variable en memoria.

Estructuras contiguas o físicas

Son aquellas que, al representarse en el hardware de la computadora, lo hacen situando sus datos en áreas adyacentes de memoria. Un dato en una estructura contigua se localiza directamente calculando su posición relativa al principio del área de memoria que lo contiene. Los datos se relacionan por su vecindad o por su posición relativa dentro de la estructura.

Figura 3. En el almacenamiento de una matriz, la información está ordenada y adyacente.

Estructuras enlazadas

Son estructuras cuyos datos no tienen por qué situarse de forma consecutiva en la memoria; estos se relacionan unos con otros mediante **punteros**. Es un tipo de dato que sirve para apuntar hacia otro dato y, así, determinar cuál es el siguiente de la estructura.

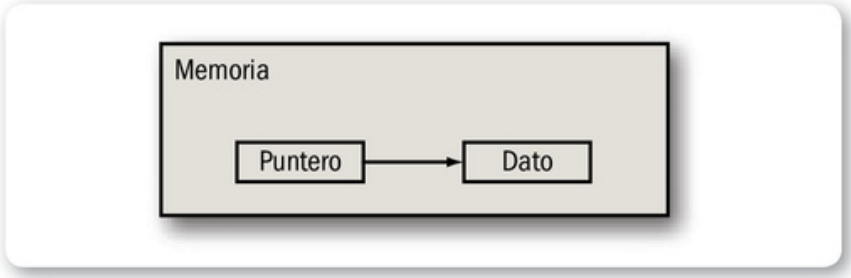


Figura 4. La localización de un dato no es inmediata, sino que se produce por la relación de datos que establece el puntero.

Ahora que sabemos categorizar algunas estructuras, al momento de desarrollar, debemos considerar cuál es la identificación de los datos que vamos a necesitar y, luego, crear la estructura que emplearemos.

Por ejemplo: si desarrollamos una aplicación para crear usuarios de un videojuego, tendremos que almacenar **código**, **nombre** y **contraseña**, dentro de una estructura que llamaremos **usuario**.

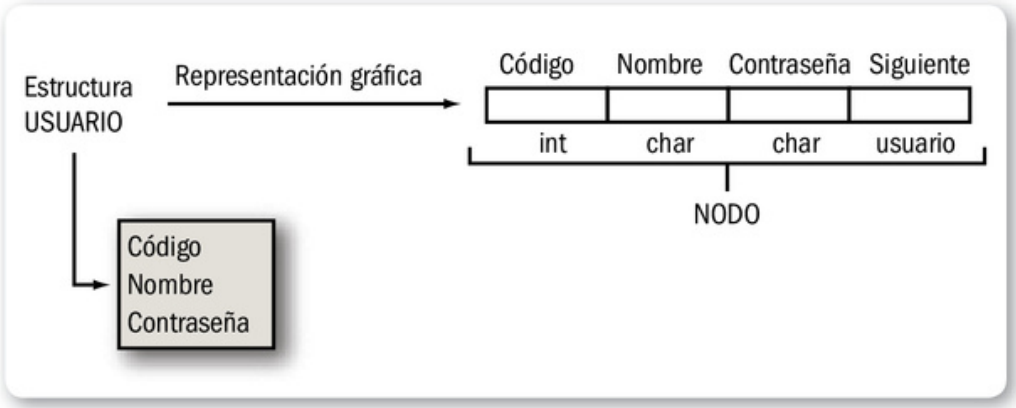


Figura 5. Este ejemplo demuestra la y representación de una estructura de datos simple.

Como podemos observar en la **Figura 5**, la representación gráfica es un esquema que podemos crear de la estructura. Esta representación no afecta a la programación, sino que es muy útil para simular los datos de la estructura que vamos a crear. Por ejemplo, otra manera de graficar es la que se muestra en la **Figura 6**.

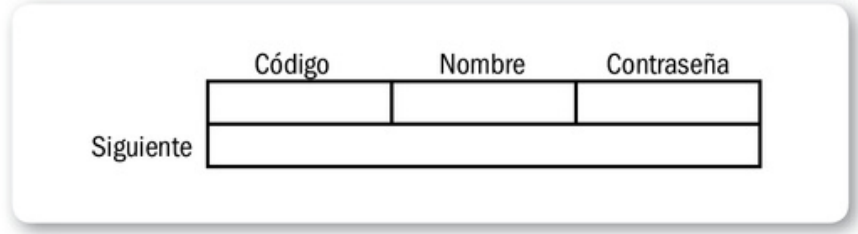


Figura 6. Este ejemplo muestra otra forma de representar gráficamente un nodo.

DEFINICIÓN DE TAD

Un Tipo Abstracto de Datos es un conjunto de valores y de operaciones definidas mediante una especificación independiente de cualquier representación. Es importante tener en cuenta que la manipulación de un TAD solo depende de su especificación, nunca de su implementación.

Estructuras dinámicas y estáticas

A lo largo de nuestro trabajo, hemos utilizado estructuras **estáticas** que representan un espacio físico de la memoria principal (variables y arreglos); aunque no estuviesen utilizadas, están ocupando un espacio "vacío" hasta que carguemos información sobre ellas. También debemos tener en cuenta que, si una estructura estática se completa, no podremos redimensionar su tamaño para agregarle más información. Esto sucede a partir del tamaño máximo de las estructuras que hayamos establecido previamente.

Por el contrario, las estructuras **dinámicas** nos permiten ir utilizando la memoria principal a medida que la vayamos necesitando. De esta forma, podemos ir creando todas las estructuras que precisemos sin tener que especificar un tamaño determinado.

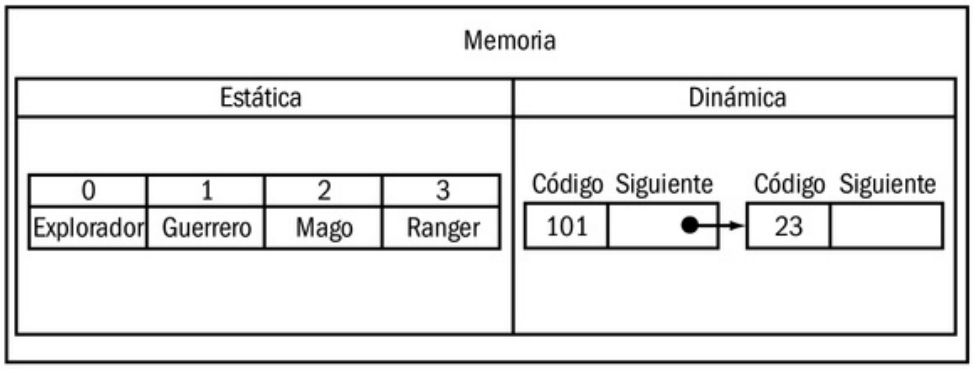


Figura 7. A la izquierda se representa una estructura estática secuencial, y a la derecha, una estructura dinámica que puede crecer.

Como podemos ver en la **Figura 7**, el vector es aquel que nos permite almacenar cuatro elementos, de modo que no podremos almacenar un quinto. Es la memoria dinámica la que, por su parte, puede ir aumentando la cantidad de nodos, siempre y cuando haya espacio suficiente en ella.

También debemos destacar que no es posible quitar elementos definidos previamente de la memoria estática. En cambio, en el caso de la memoria dinámica, sí podemos deshacernos de todos aquellos nodos que no sean necesarios.

Retomando las características de las estructuras complejas, en la **Tabla 1** encontramos la siguiente categorización, que ampliaremos en detalle más adelante.

COMPLEJAS	
▼ LINEALES	▼ NO LINEALES
Lista	Árbol
Pila	Grafo
Cola	*

Tabla 1. Estructuras complejas.

Estructuras dinámicas y punteros

Hasta aquí hemos trabajado con variables simbólicas que poseen una relación directa entre su nombre y ubicación durante toda la ejecución del desarrollo. Cabe destacar que el contenido de una posición de memoria asociada con una variable puede cambiar durante la ejecución, y modificar así el valor asignado.

Definimos las estructuras dinámicas como aquellas que nos permiten adquirir posiciones de memoria adicionales, a medida que lo vayamos necesitando en el desarrollo, y liberando dichos espacios cuando no sean requeridos. Las mismas se representan con la ayuda de un tipo de dato llamado **puntero**, que indica la posición de memoria ocupada por otro dato. Podríamos representarlo como una flecha que señale al dato en cuestión.

DINAMISMO

En las estructuras complejas, la longitud no es fija en memoria, sino que va aumentando o decreciendo durante la ejecución del programa, de acuerdo con los datos que se añaden o eliminan. Esto nos permite asignar y liberar dinámicamente la memoria principal.

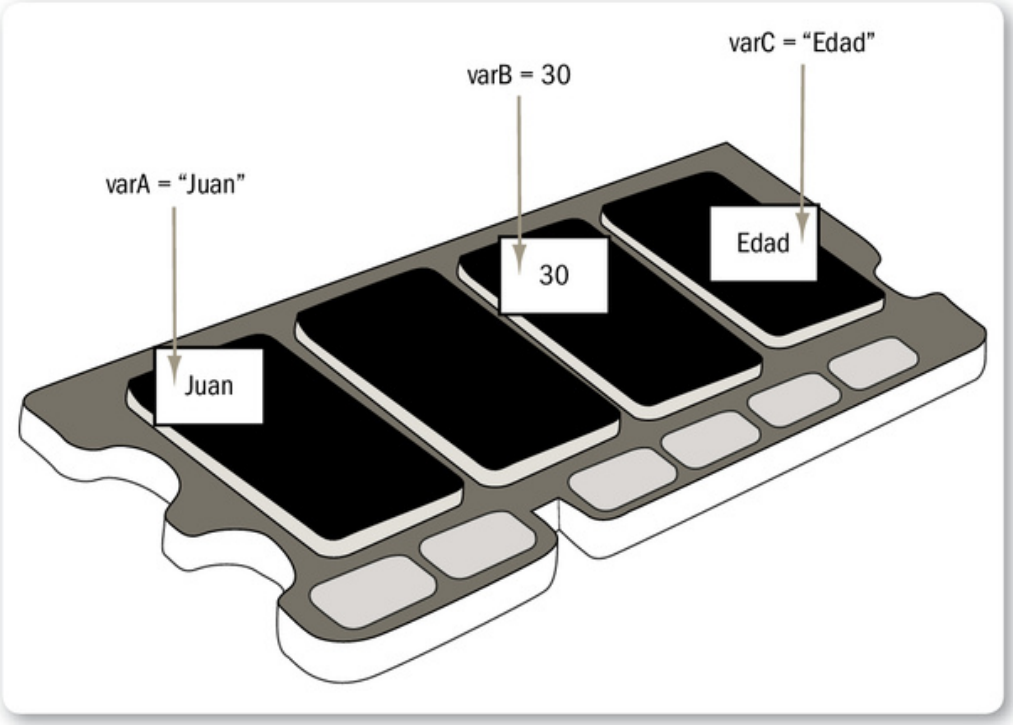


Figura 8. Representación de cómo se crean nodos en memoria RAM, viendo su nombre y asignación.

Como podemos ver en la **Figura 8**, aquí hemos creado tres nuevos elementos en memoria: Juan, 30 y Edad. Si queremos identificar cuál es su posición física exacta, debemos crear variables de referencia que estén asociadas al dato que nosotros vamos a almacenar.

En pseudocódigo, la sintaxis para la creación de estas referencias es:
Variable varA tipo **nodo**
Variable varB tipo **nodo**
Variable varC tipo **nodo**

ABSTRACTO

Las estructuras complejas no poseen un operador constructor, es decir, no se definen previamente. Permiten manejar los datos, tales como: cola, pila, lista enlazada, árbol o grafo, en función de las características particulares de cada una y de las necesidades del desarrollo.

Como podemos ver, lo que definimos como objeto para identificar los elementos y asociarlos es de tipo **nodo**. Para relacionar la información que grabamos en memoria de manera dinámica o enlazarlos, debemos tener un espacio o apuntador que nos indique dónde se está grabando el próximo dato de la estructura. Para eso, hay que tener una estructura heterogénea que almacene la información.

En este caso, deseamos almacenar: "Juan", "Edad" y "30". Veamos cómo debería ser su relación en la **Figura 9**.

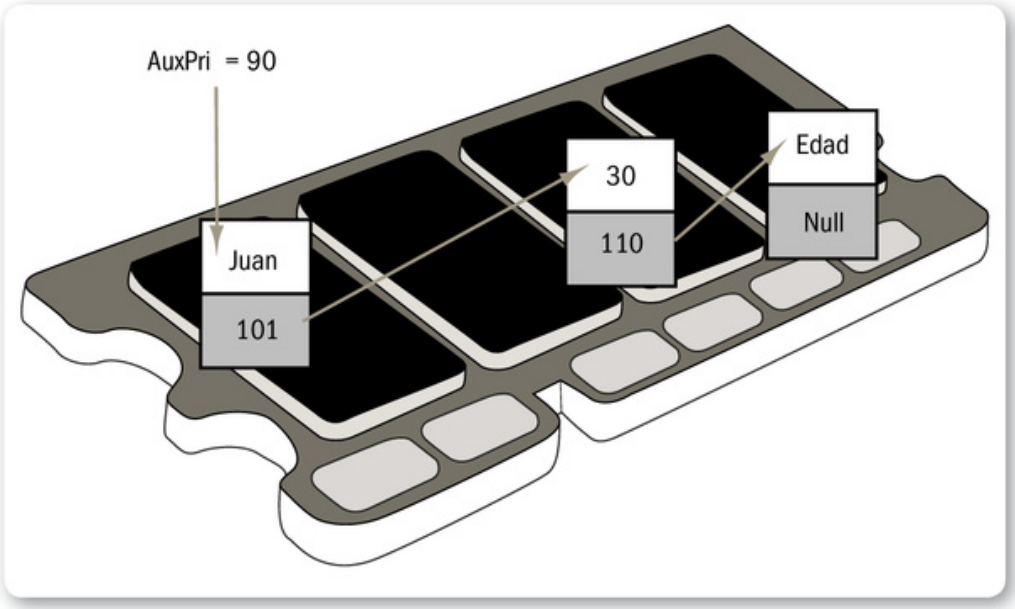


Figura 9. Representación de cómo sería una relación de nodos en direcciones de memoria RAM.

Como podemos observar, es necesario saber dónde estará almacenado el primer nodo. Para hacerlo, crearemos una variable auxiliar "AuxPri", que guarda la dirección de memoria donde está almacenado. Es

PUNTEROS

Los punteros proporcionan los enlaces entre los elementos, permitiendo que, durante la ejecución del programa, las estructuras dinámicas cambien sus tamaños. Los encontramos en los nodos, ya que forman parte de su estructura de al menos dos campos.

EL ÚLTIMO ELEMENTO DE NODOS NOS INDICARÁ EL FINAL DE LA ESTRUCTURA

importante tener en cuenta que las direcciones de memoria son más complejas, por eso en el gráfico vemos un ejemplo metafórico.

Con AuxPri podemos ubicarnos sobre el primer nodo, donde tenemos "Juan"; a su vez, este nos indica la dirección del siguiente dato con su **puntero**, que es de valor 101 y nos lleva a "30". Este, por su parte, nos indica a "Edad", que es donde terminaría nuestro recorrido de nodos debido a que su **puntero** es **Null**. Cuando lleguemos al último elemento de nodos, su contenido será **Null** en el puntero, y esto nos indicará el final de la estructura.

Veamos la definición de un nodo en pseudocódigo, donde almacenaremos una lista de valores numéricos:

```
Estructura Nodo
        Variable Dato tipo numérica
        Variable Siguiente tipo nodo
Fin estructura

Variable Primero tipo nodo
Variable Ultimo tipo nodo
```

Veamos la definición de nodo en C++:

```
struct Nodo
{
        int Dato;
        struct Nodo *sig;
};
* (Asterisco) nos indica que es una variable para guardar una dirección de
        memoria.

void main()
{
        struct Nodo *primero, *ultimo;
};
```

La estructura de **Nodo** se encuentra integrada por dos variables: una donde se almacena el dato, y otra donde se guarda la dirección física de memoria para el próximo dato.

Todas las estructuras dinámicas necesitan, por lo menos, un puntero auxiliar que indique dónde está almacenado el primer nodo; también podemos definir un auxiliar que nos indique el último nodo.

Al terminar la declaración, el lenguaje que estemos utilizando asignará **Null** a las declaraciones, y quedará de la siguiente manera:

```
Primero = Null
Ultimo = Null

Nodo.Dato = 0
Nodo.Siguiente = Null
```

Crear y eliminar nodos

Para la creación de nodos, podemos emplear en C++ un operador llamado **new**. A su vez, este puede retornar dos valores distintos: **Null** o una dirección de memoria. El operador retornará **Null** cuando el operador intente crear el nodo en un espacio de memoria y no encuentre el lugar suficiente donde hacerlo; de lo contrario, devolverá la dirección de memoria donde será almacenado.

Por ejemplo, continuando con la codificación anterior, veamos el siguiente código fuente:

```
struct Nodo *nuevo;
        nuevo = new Nodo;

//se recomienda corroborar que el nodo se haya creado en memoria, para ello:

        If(nuevo==Null)
                cout<<"No hay espacio en memoria";
else
{
//instrucciones que vayamos a trabajar en el nodo.
}
```

El operador **delete** sirve para quitar un nodo de la memoria. En ese caso, debemos indicar el puntero del nodo que deseamos eliminar.

Por ejemplo:

```
delete primero;
```

Cuando creamos un nodo desde **nuevo**, este almacenará la dirección asignada para la estructura, pero no hará referencia a los campos que contiene. En la **Figura 10**, vemos un ejemplo.

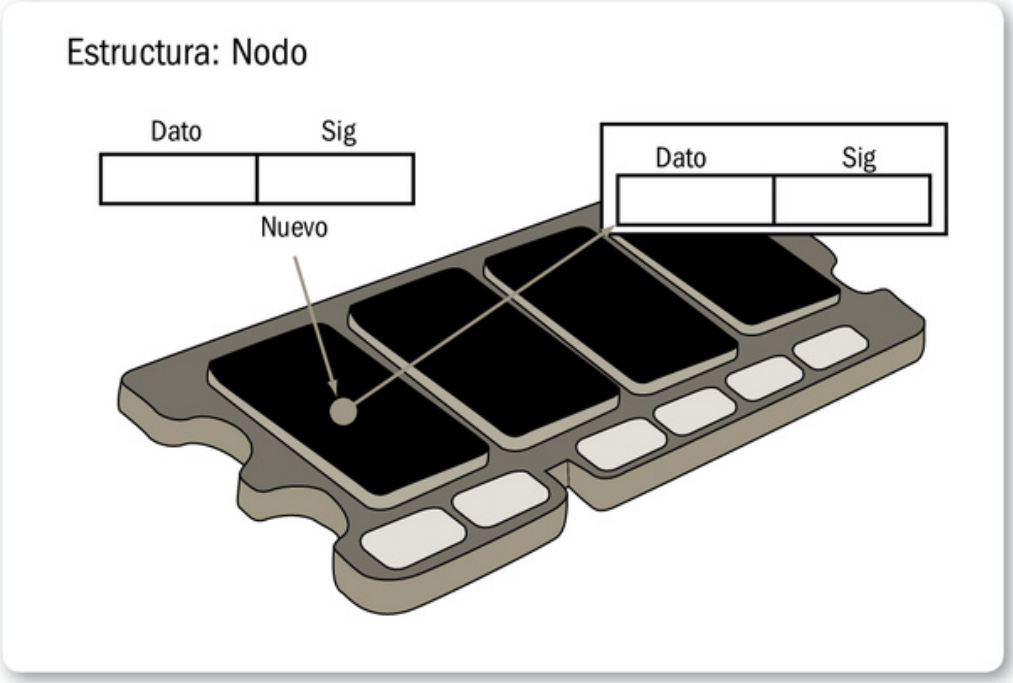


Figura 10. Representación de cómo sería la declaración de un nodo nuevo en memoria RAM.

Continuando con el código anterior, veamos ahora cómo podemos asignar datos a los campos del nodo dentro del **else** de nuestro condicional. Para eso escribiremos:

```
else
{
```

```
        nuevo -> dato = 3;
        nuevo -> sig = Null;
}
```

Trabajar con punteros

Ahora analicemos qué sucede si deseamos ordenar nodos o intercambiar sus espacios de memoria. Para eso, primero revisemos los siguientes ejemplos:

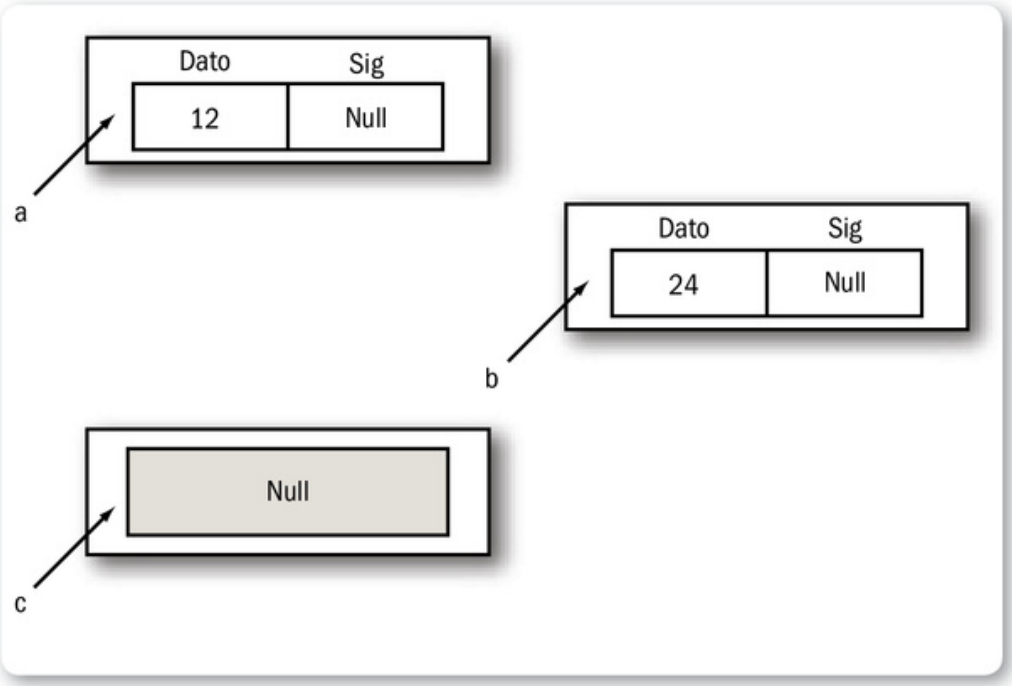


Figura 11. Representación de cómo estarían declarados los nodos sueltos en memoria.

Ejemplo 1:

- Punteros: a, b y c
- Nodos: 2
- Le pedimos que ordene los valores de mayor a menor y en memoria.

Si necesitamos cambiar el espacio de memoria del nodo con el valor 24, en lugar de 12 y viceversa, debemos hacer lo siguiente:

```
        struct Nodo *a;
        struct Nodo *b;
        struct Nodo *c;
  //declaración de los punteros a utilizar
```

Propuesta 1: Asignar el valor del puntero **a** al **b**.

```
        b=a;
```

Y ahora debemos asignar la dirección del nodo con el valor 24. Pero perdimos el espacio de memoria que nos indicaba la posición de dicho nodo, ya que **b** tiene el valor de **a**.

Por lo tanto, esta propuesta no es útil.

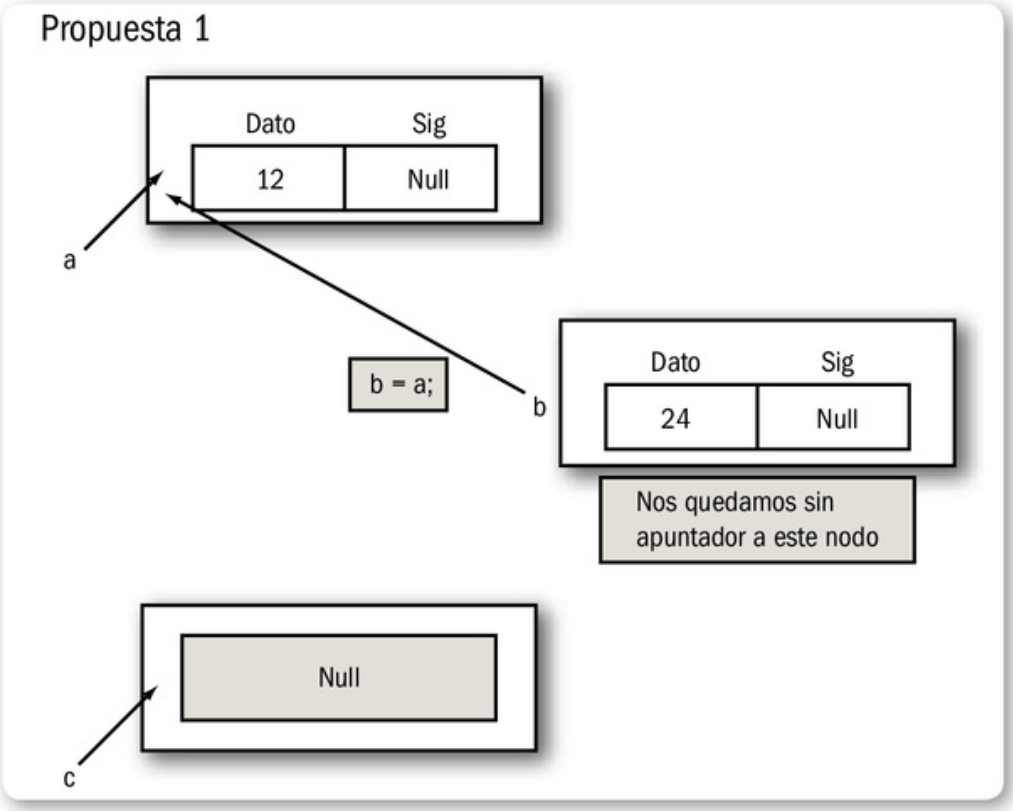


Figura 12. Representación si ejecutamos la Propuesta 1, relacionando los punteros y nodos.

Propuesta 2: Utilicemos el puntero que tiene valor nulo. Por ejemplo:

```
c=b;
```

De esta manera, considerando que el espacio de memoria del nodo correspondiente al valor 24 está almacenado en el apuntador c, podemos realizar las siguientes asignaciones:

```
b=a;
a=c;
```

El ejemplo quedaría como se observa en la **Figura 13**.

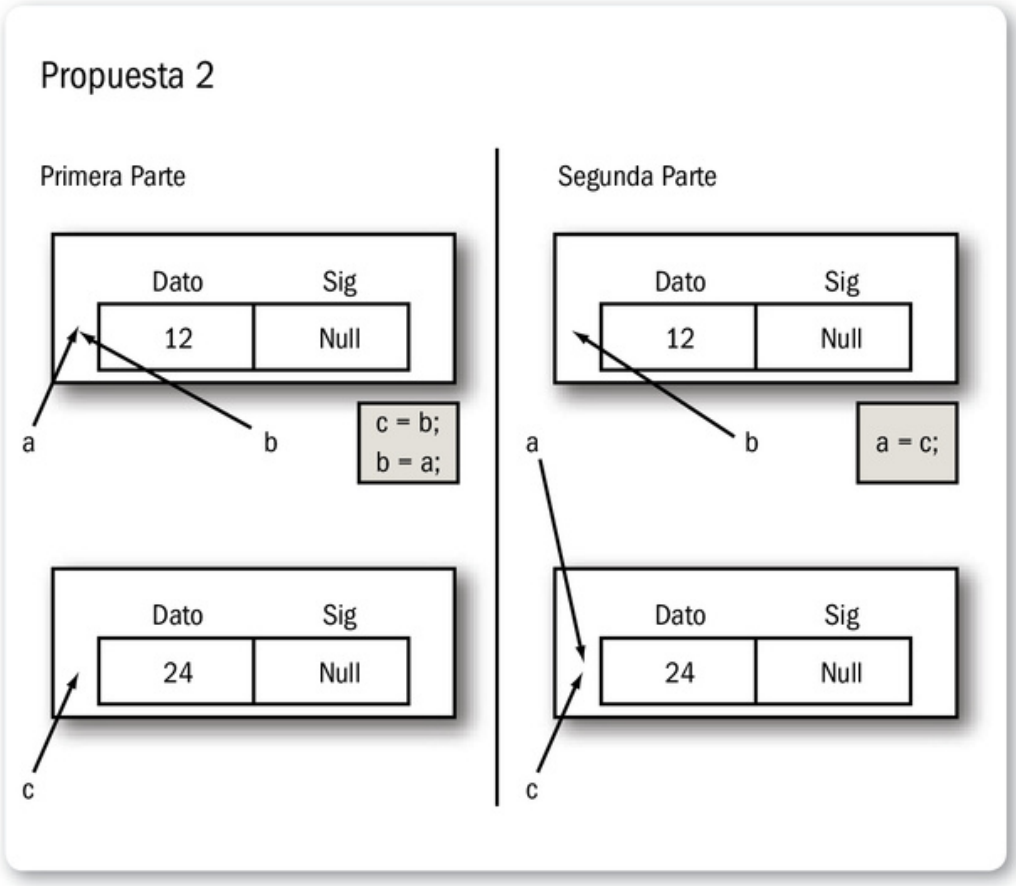


Figura 13. Representación si ejecutamos la Propuesta 2, relacionando los punteros y nodos.

Como podemos observar en las imágenes, la propuesta 1 no nos será útil para intercambiar los espacios de memoria entre los nodos. En cambio, la 2 muestra la forma apropiada de hacer uso de los **punteros** y realizar la tarea de intercambio.

Ejemplo 2:

- Nodos: 3
- Punteros: a y c.
- El nodo con el valor 1 (puntero "a") tiene un apuntador al nodo con el valor 2.
- El nodo con el valor 3 (puntero "c") no tiene ningún enlace.

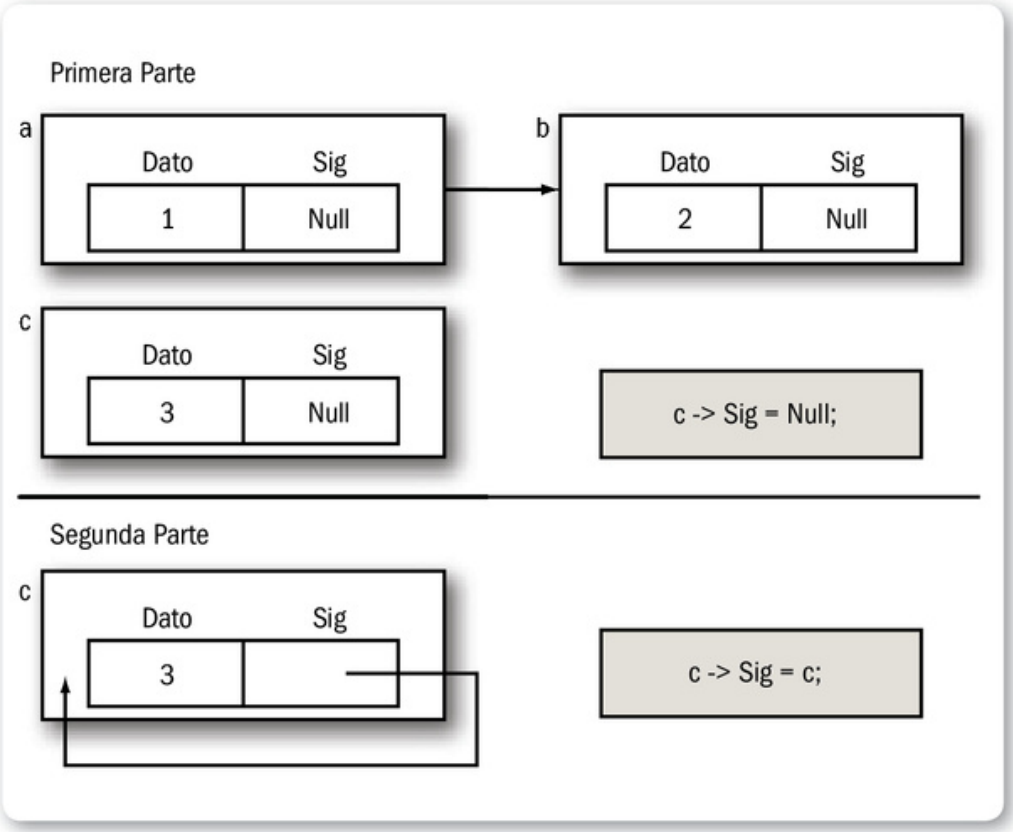


Figura 14. Representación y código fuente de cómo utilizamos punteros para operar nodos.

En la primera parte vemos que el puntero **c** indica el elemento **Sig** y le asigna nulo. En la segunda, vemos que el puntero **c** indica al elemento **Sig** la posición de **c**. Esto quiere decir que guarda la dirección

de memoria de **c**, y si revisamos lo que es recursividad, encontraremos una similitud en esta asignación a dicho concepto.

Estas asignaciones se pueden leer de diferentes formas y todas son correctas:

```
c->Sig=NULL;
```

- Al apuntador **c** en el campo **Sig** asignamos nulo.
- En el campo **Sig** del apuntador **c** asignamos nulo.
- En el apuntador **c** hay un campo llamado **Sig**, donde asignamos nulo.

Ahora, si deseamos que el nodo del puntero **c** en su campo **Sig** se enlace con el nodo del puntero **a**, el código deberá ser el siguiente:

```
c->Sig=a;
```

Por último, también podemos tomar campos del nodo y asignarlos al puntero de la siguiente manera:

```
c->Sig=a->Sig;
```

Veamos el gráfico de la **Figura 15** para entender estos ejemplos.

De esta manera, hemos visto una breve representación de cómo se enlazan los datos en memoria y qué es importante tener en cuenta para ir aplicando las estructuras dinámicas. A continuación, distinguiremos las formas de manejar la memoria dinámica con respecto a las estructuras complejas que estudiaremos a lo largo del capítulo.

RECURSIVIDAD

Las estructuras complejas son consideradas recursivas, a partir de dos aspectos:

-Algoritmia: por la complejidad de la estructura, requiere de algoritmos recursivos.

-Definición: cada nodo tiene definido un dato simple en el que guarda la referencia a un nodo.

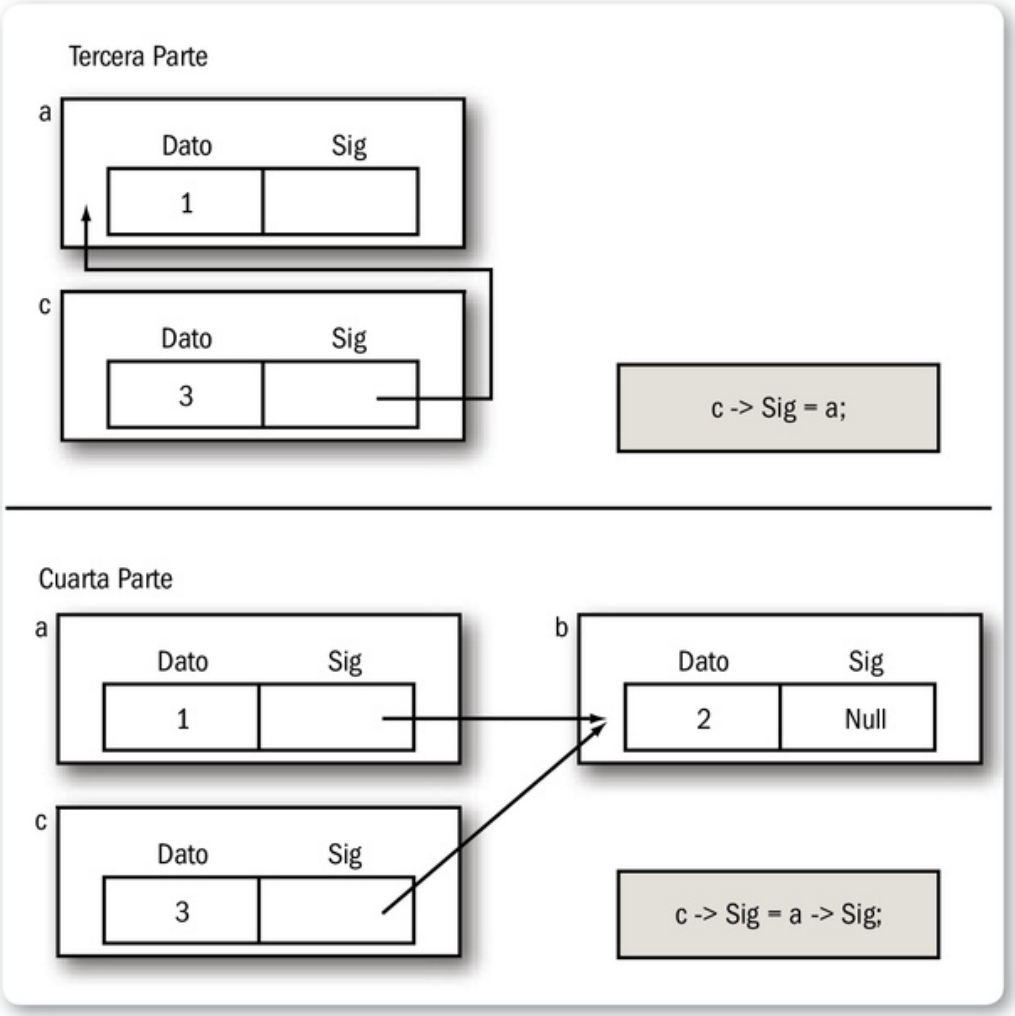


Figura 15. Representación de cómo sería asignar punteros de un nodo a otro.

Lista

Si tuviésemos que dar una definición amplia acerca de qué significa una lista dentro de una estructura de datos, diríamos que se trata de un conjunto de datos de un mismo tipo (simple o estructurado), donde cada elemento tiene un único predecesor (excepto el primero) y un único sucesor (excepto el último). También sería importante agregar que el número de elementos es siempre variable.

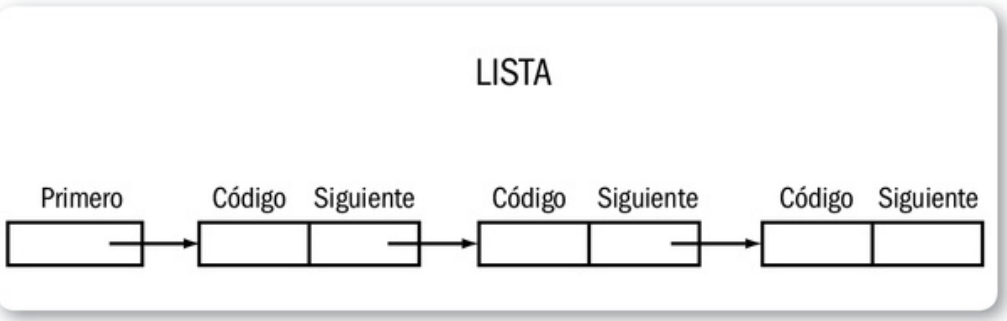


Figura 16. Representación de cómo sería una lista y la asociación entre sus nodos.

Se distinguen dos tipos de listas: **contiguas** y **lineales**. Las primeras son estructuras intermedias entre las estáticas y las dinámicas, ya que sus datos se almacenan en la memoria en posiciones sucesivas y se procesan como arreglos (vectores/matrices). Lo bueno de trabajar con esta disposición secuencial, el acceso a cualquier elemento de la lista y la adición de nuevos elementos es sencillo, siempre que haya espacio suficiente para hacerlo.

Para que una lista contigua pueda variar de tamaño (y, por lo tanto, de la impresión de una estructura dinámica), es necesario definir un arreglo dimensionado por tamaño suficiente para que pueda contener todos los posibles elementos de la lista. Cuando hablamos de una lista contigua, nos referimos a un arreglo que tiene posiciones libres por delante y detrás, y cuyo índice hace de puntero.

Por otro lado, tenemos las listas lineales –llamados lista, pila y cola–, que son las que veremos en los siguientes párrafos.

Listas enlazadas

Estas listas se forman por conjuntos de nodos, en donde cada elemento contiene un puntero con la posición o dirección del siguiente elemento de la lista, es decir, su enlace. Como vimos anteriormente, los nodos están compuesto por dos campos: uno donde se almacena información y otro donde estará la posición del siguiente nodo. Con esta organización de datos, es evidente que no será necesario que

los elementos de la lista estén almacenados en posiciones físicas adyacentes para estar relacionados entre sí, ya que el puntero indica la posición del dato siguiente en la lista. En la **Figura 9** podremos observar un claro ejemplo de esto.

En resumen, podemos destacar que una lista enlazada es aquella que está definida por una estructura de tipo nodo y un puntero que indica el primer elemento, a partir del cual se puede acceder a cualquier otro elemento de la agrupación.

Veamos en la **Figura 17** otro ejemplo acerca de cómo quedaría enlazada una lista con los números 1, 3, 5, 7.

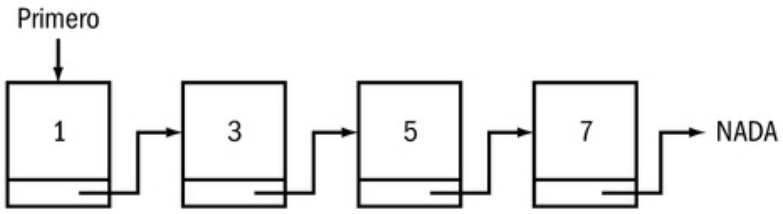


Figura 17. Representación de una lista enlazada, que muestra sus punteros y contiene datos de tipo números.

Primero sería el puntero el que señala el primer elemento de la lista. Mediante el puntero situado en cada uno de los nodos, se puede acceder al siguiente elemento desde cualquiera de ellos. La ventaja del uso de nodos es que podemos agregar y retirar información en cualquier ubicación de la lista.

Antes de avanzar a la creación de listas enlazadas, veamos otros ejemplos sobre ellas, en la **Figura 18**.

HISTORIA

Las listas enlazadas fueron desarrolladas en 1955-56 por Cliff Shaw y Herbert Simon en RAND Corporation, como la principal estructura de datos para su lenguaje de procesamiento de la información (IPL). IPL fue usado para desarrollar varios programas relacionados con la inteligencia artificial.

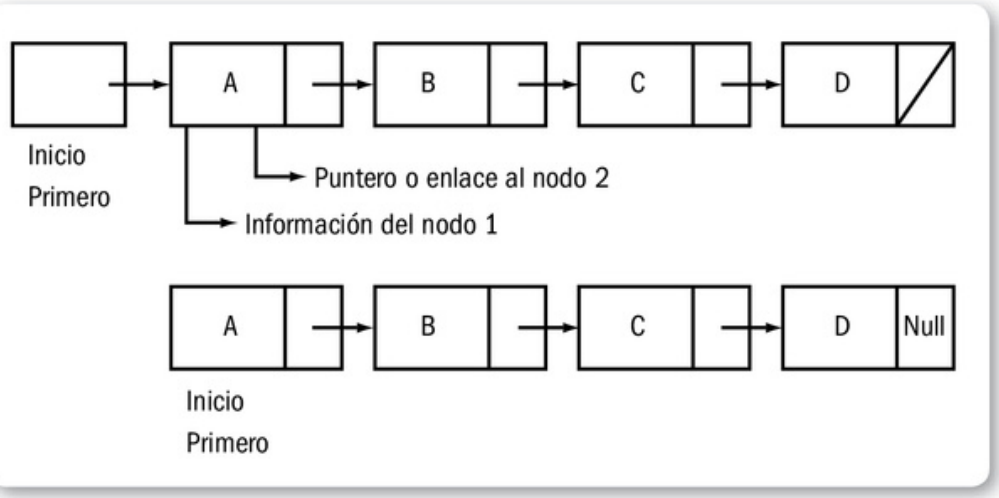


Figura 18. Aquí podemos ver dos listas enlazadas y la forma de representar el valor nulo.

Como podemos ver, también podemos simbolizar el final de la lista con una barra cruzada "\".

Creación

Para implementar listas enlazadas, debemos tener en cuenta el lenguaje de programación que utilizamos, ya que no todos soportan el puntero como tipo de dato. En este caso, en que empleamos C++ y Visual Basic, podemos utilizarlos tranquilamente.

La alternativa al uso de punteros es recurrir a vectores paralelos en los que se almacenan los datos correspondientes a los campos utilizados en un nodo. Con una variable que apunte al índice que contiene la cabecera de la lista (que nosotros utilizamos como puntero **primero**), podemos imaginar un esquema como se muestra en la **Figura 19**.

LENGUAJE LISP

Lisp es una familia de lenguajes de programación de computadora de tipo multiparadigma con una sintaxis completamente entre paréntesis. Fue creado en 1958 y considerado el principal procesador de listas. Una de las mayores estructuras de datos de LISP es la lista enlazada.

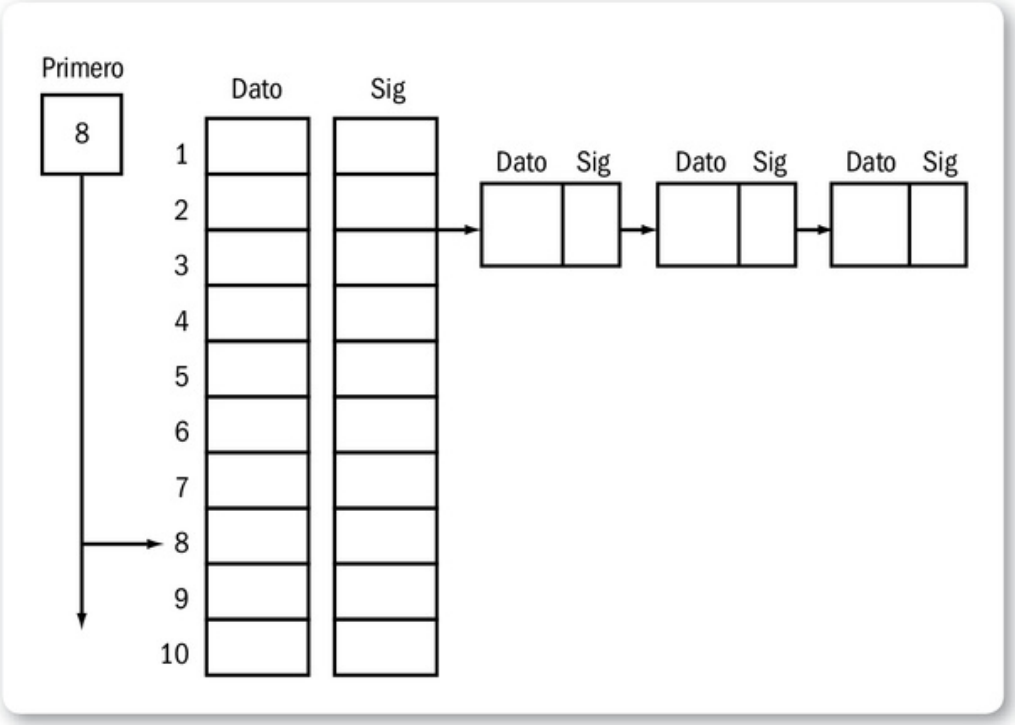


Figura 19. Representación de un esquema de estructuras que podemos utilizar con la creación de tipos de datos diferentes.

En la imagen vemos una estructura de vector en la parte izquierda, y una de nodos en la derecha.

Una vez definida la estructura de nodos, para crear una lista necesitamos llenar un primer nodo con información que corresponda al tipo de elemento y que el enlace de este contenga el valor nulo. No nos olvidemos de definir un puntero externo con el valor de la dirección del nodo inicial. A partir de él, **primero**, la lista puede modificarse, crecer o disminuir, incluir más nodos y borrar otros.

ABSTRACCIÓN

La abstracción es la operación mediante la cual formamos conocimiento conceptual común a un conjunto de entidades, separando de ellos los datos contingentes e individuales para atender a lo que los constituye esencialmente. En definitiva, se trata de aislar mentalmente o considerar por separado las cualidades de un objeto. Esta es una de las tareas que llevaremos a cabo para crear datos.

Tengamos en cuenta que podemos utilizar un estándar para la sintaxis algorítmica, por ejemplo:

Primero: se trata de un puntero externo correspondiente al primer nodo de una lista enlazada.

P: es un puntero a un nodo cualquiera de la lista.

P->Nodo: el nodo apuntado por P.

p->Dato: campo de datos del nodo apuntado por P.

p->Sig: campo puntero del nodo apuntado por P (que generalmente apuntará al nodo siguiente).

Recorrido

Para recorrer estructuras estáticas como los arreglos (vector/matriz) utilizábamos una repetitiva que recorría de inicio a fin. En el caso de las listas, al no contar con un índice ordenado o datos sucesivos, podemos pensar que no es sencillo acceder de manera directa o aleatoria a los nodos. Esto se debe a que tenemos que acceder al primero mediante un puntero externo, al segundo, etc. Para resolver esta situación, debemos utilizar una variable puntero auxiliar, que generalmente denominamos **X**. Su función será apuntar en cada momento al nodo procesado, simplemente, asignando: **x = x->Sig**, y esto guardará en **x** el campo puntero del nodo.

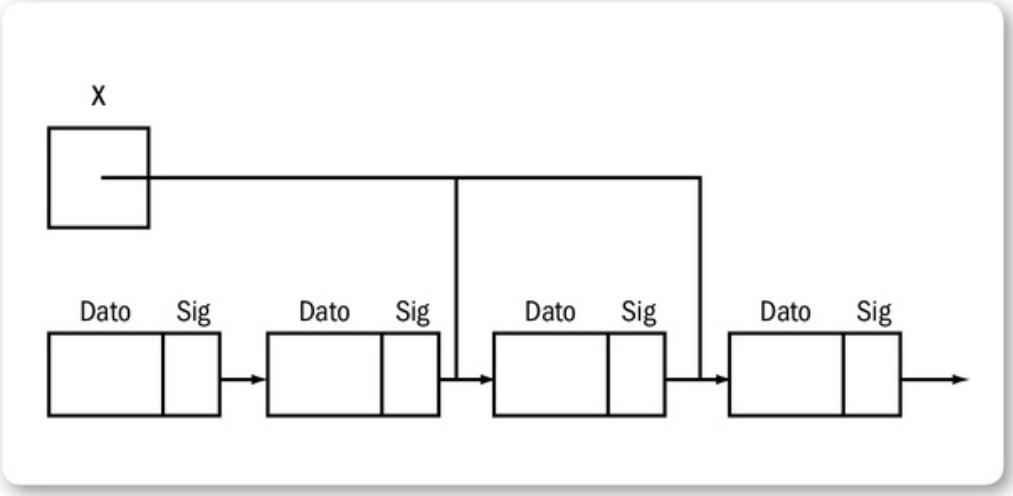


Figura 20. Representación del ejemplo anterior, donde utilizamos la asignación del puntero **x=sig**.

Veamos un ejemplo en pseudocódigo y en C++, teniendo en cuenta que el recorrido de una lista es posible si utilizamos el puntero temporal **P**. Esto nos permitirá leer la lista completa:

PSEUDOCÓDIGO	C++

Inicio
P<--Primero
Mientras P<>Nulo hacer
 	
Escribir P.Dato
 	
P=P.Sig
Fin Mientras
Fin
	void main()
	{
	//aquí debe ir las instrucciones para crear la lista y punteros
	
	P = primero;
	
	while P!=NULL
	{
	cout << P->Sig;
	p = p->Sig;

El puntero **P** contiene el valor del primer elemento de la lista. En el bucle recorremos toda la lista (de inicio a fin) hasta que encuentre un nodo cuyo dato **siguiente** sea nulo. De esta forma, nos daremos cuenta de que corresponde al último nodo de la lista, y sale del bucle.

Veamos otro ejemplo donde tengamos una lista de valores numéricos con extensión indefinida. Para eso, debemos contar todos los elementos existentes en la lista. Veamos el código fuente:

PSEUDOCÓDIGO	C++

Inicio
Contador<--0
P<--Primero
Mientras P<>Nulo hacer
 	
Contador <-- Contador +1
 	
P=P.Sig
Fin Mientras
ESCRIBIR Contador
	void main()
	{
	//aquí debe ir las instrucciones para crear la lista, punteros y variables
	
	contador=0;
	P = primero;
	
	while P!=NULL

```
Fin
```

```
{
        contador++;

        p = p->Sig;
}
cout<< "Cantidad de elementos "
+ contador
}
```

De esta manera, podremos recorrer sin problemas las listas enlazadas de inicio a fin. Ahora observemos las **Figuras 16**, **17** y **18** para destacar cómo sería el recorrido de las listas.

Búsqueda

La búsqueda en una lista enlazada debe hacerse mediante un recorrido elemento por elemento, hasta encontrar el deseado o llegar al final de la lista (sea nulo). También es importante tener en cuenta que podemos encontrar listas ordenadas y desordenadas; por lo tanto, la búsqueda se puede modificar dependiendo de este aspecto. Teniendo en cuenta esto, veamos los siguientes ejemplos:

Nos encontramos con una lista enlazada cuyo primer nodo está apuntado por **PRIMERO**, el siguiente procedimiento busca un elemento **x** obteniendo un puntero **POS** (posición) que lo apunta.

PSEUDOCÓDIGO

Procedimiento BusquedaDesordenada(Primero, datoBuscado, POS)

//en el procedimiento tenemos los parámetros del puntero del primer nodo, el valor buscado y el lugar del puntero que ocupa POS.

```
Inicio
        P<--Primero
        POS<--NULO

        Mientras P<>Nulo hacer
```

```
Si datoBuscado=P.dato entonces
        POS <--P

        P<--NULO
Sino

        P<--P.Sig
Fin si
        Fin Mientras
        Si POS=NULO entonces

        Escribir "No se encontró el dato buscado"
Fin si
Fin
```

C++

```
  void BusquedaDesordenada(Primero, datoBuscado, POS)
{
 P=Primero;
 POS=NULL;

 while(P!=NULL)
 {
  if(datoBuscado=P->Dato)
  {
   POS=P;
```

ENCAPSULADO

El concepto encapsulado deriva de un proceso de abstracción y consiste en ocultar la implementación de las operaciones que manipulan los objetos, ofreciendo únicamente una interfaz que permita realizar operaciones, funciones de acceso y variables de entrada/salida. Un ejemplo de ello sería la representación de enteros en diferentes notaciones int, int32 o Long.

```
    P=NULL;
  }
  else
  {
    P=P->Sig;
  }
 }
 if(POS==NULL)
 {
     cout<<"No se encontró el dato buscado";
 }
}
```

Ahora, suponiendo que estamos trabajando en una lista ordenada en forma ascendente, vamos a revisar la codificación para el siguiente algoritmo de recorrido:

PSEUDOCÓDIGO

Procedimiento BusquedaOrdenada(Primero, datoBuscado, POS)

//el procedimiento tenemos los parámetros del puntero del primer nodo, el valor buscado y el lugar del puntero que ocupa POS.

```
 Inicio
        P<--Primero
        POS<--NULO

        Mientras P<>Nulo hacer

        Si P.dato<datoBuscado entonces

        P<--P.Sig
  Sino
   si datoBuscado = P.dato entonces

        POS <--P
```

```
P<--NULO
 Fin si
Fin si
        Fin Mientras
        Si POS=NULO entonces

        Escribir "No se encontró el dato buscado"
Fin si
Fin
```

Podemos destacar la diferencia entre este código **BusquedaOrdenada** y el anterior, en la sentencia condicional **Si P.dato<datoBuscado entonces**, ya que al estar ordenados sus elemento y ser numéricos, podemos preguntar si el dato en la lista es menor al dato que estamos buscando.

Inserción de elementos

Cuando necesitamos **insertar** o **borrar** un nodo de una lista, es importante tener en cuenta si esta se encuentra ordenada o no. La tarea principal será, simplemente, modificar los punteros de esta estructura. Si debemos insertar un nuevo nodo, este puede ubicarse al inicio o a continuación de un nodo específico.

Cualquiera sea el tipo de inserción, siempre será necesario contar con un nodo vacío en donde almacenaremos información.

A continuación, veamos en la **Figura 21** un ejemplo sencillo de esta situación y el código fuente que podemos insertar en una lista.

MÉTODOS DE BÚSQUEDA

En la programación encontraremos distintos métodos de búsqueda, basados en algoritmos diseñados para localizar un elemento con ciertas propiedades dentro de una estructura de datos.

Por ejemplo, ubicar el registro correspondiente a cierta persona en una base de datos, o el mejor movimiento en una partida de ajedrez. Dentro de los tipos de búsquedas, podemos encontrar secuencial, secuencial indexada, binaria, de cadenas, interpolación, etc.

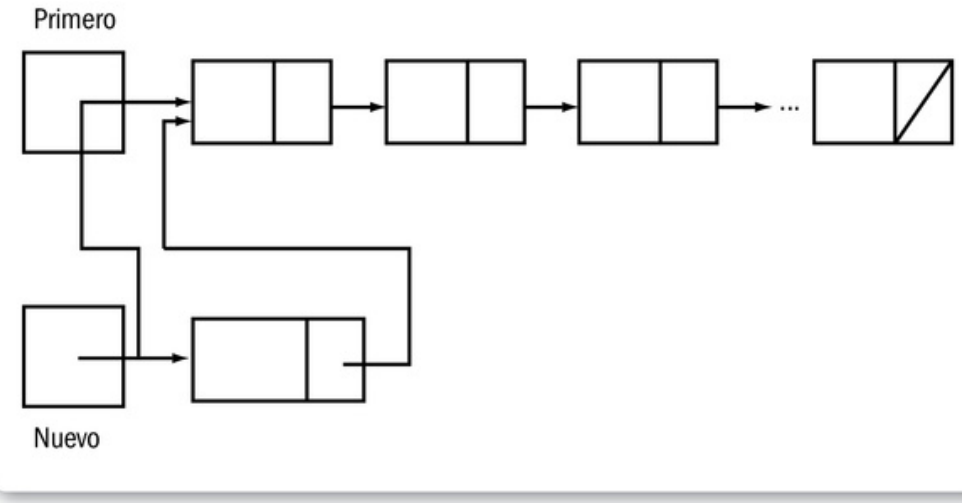


Figura 21. Representación gráfica de cómo sería la inserción en el primer nodo de la lista.

A continuación, utilizaremos como ejemplo una lista con la estructura que venimos manejando, en donde el contenido de la información será 2, 6, 7. Veremos tres escenarios de inserción:

- Nodo menor al primero.
- Nodo mayor al último.
- Nodo mayor al primero y menor al último.

Utilizaremos un auxiliar llamado **Disponible**, que obtiene un nuevo nodo si es posible; en caso contrario, dará el valor nulo. También usaremos una variable **NvaInfo** para representar el dato que deseamos insertar en la lista.

PSEUDOCÓDIGO: Nodo menor al primero

MODULARIDAD EN LA PROGRAMACIÓN

La función de la modularidad es descomponer un programa en un pequeño número de abstracciones. Estas partes tienen la característica de ser independientes unas de otras, pero fáciles de conectar entre sí. Un módulo se caracteriza principalmente por su implementación, y su programación sigue un criterio de ocultación de la información. Gracias a este modelo, el sistema se asegura de mostrar solo aquella información que sea necesario presentar.

```
Inicio

        NUEVO<--Disponible
        Si NUEVO=NULO entonces

        Escribir "Desbordamiento de memoria"
Sino

        NUEVO.dato<--NvaInfo
NUEVO.sig<--Primero
Primero<--NUEVO //el nuevo nodo es la cabecera de la lista

Fin si
Fin
```

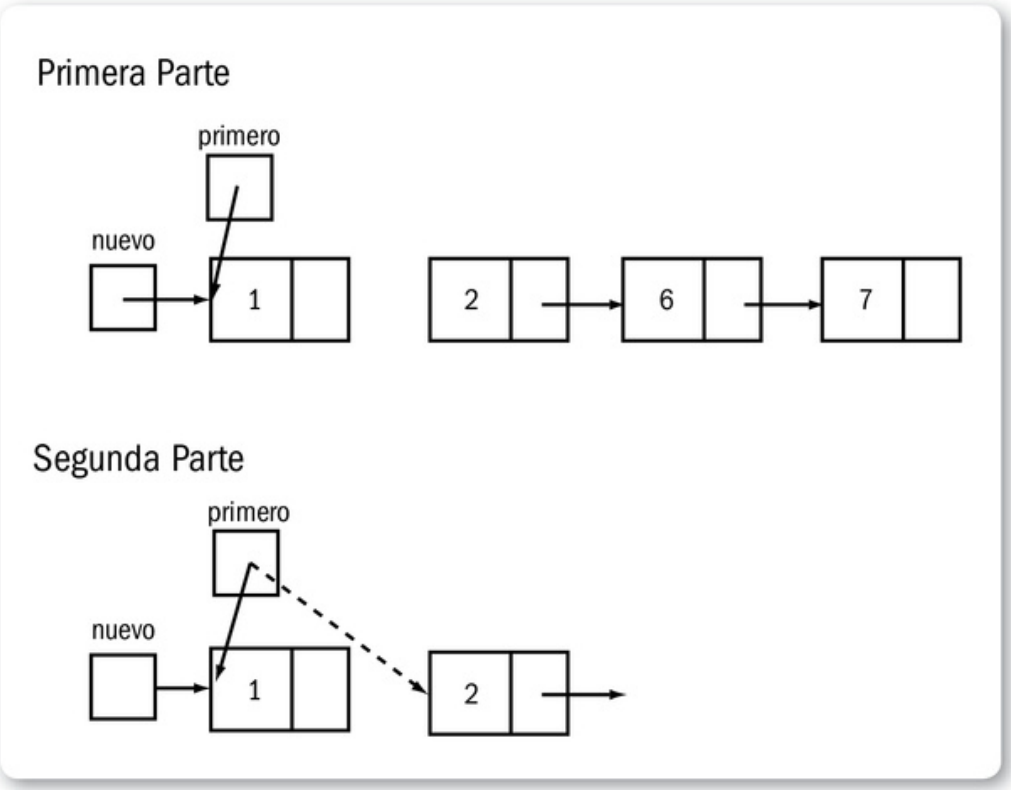


Figura 22. Aquí se representa si el valor que se ingresa es menor que el primer nodo, y cómo sería la asignación del puntero.

En caso de tener que insertar un valor entre nodos, necesitaremos utilizar los apuntadores auxiliares; como podemos apreciarlo en la Figura 23. Los nodos a insertar pueden haber sido creados, pueden existir ya en el mismo documento o se pueden importar de otro documento. En el pseudocódigo que vemos a continuación, recordemos los auxiliares P y otro nuevo Q, y 5 es el valor que queremos grabar en un nuevo nodo.

PSEUDOCÓDIGO: Nodo mayor que el primero y menor que el último

```
Inicio

        NUEVO<--Disponible
        Si NUEVO=NULO entonces
Escribir "Desbordamiento de memoria"
Sino

        NUEVO.dato<--NvaInfo

        Q<--P.Sig

        P.Sig<--NUEVO

        NUEVO.sig<--Q

Fin si
Fin
```

RUNNING TIME

El running time de un algoritmo representa el número de operaciones primitivas o de pasos a seguir que deben ejecutarse. Este va a depender de la magnitud del tamaño que tenga la entrada de información. Se trata del tiempo que tarda en ejecutarse un programa, momento en el cual el sistema operativo comienza a ejecutar sus instrucciones. Al conocer el factor de crecimiento del running time, se puede predecir cuánto tiempo tardará el algoritmo con una entrada mayor.

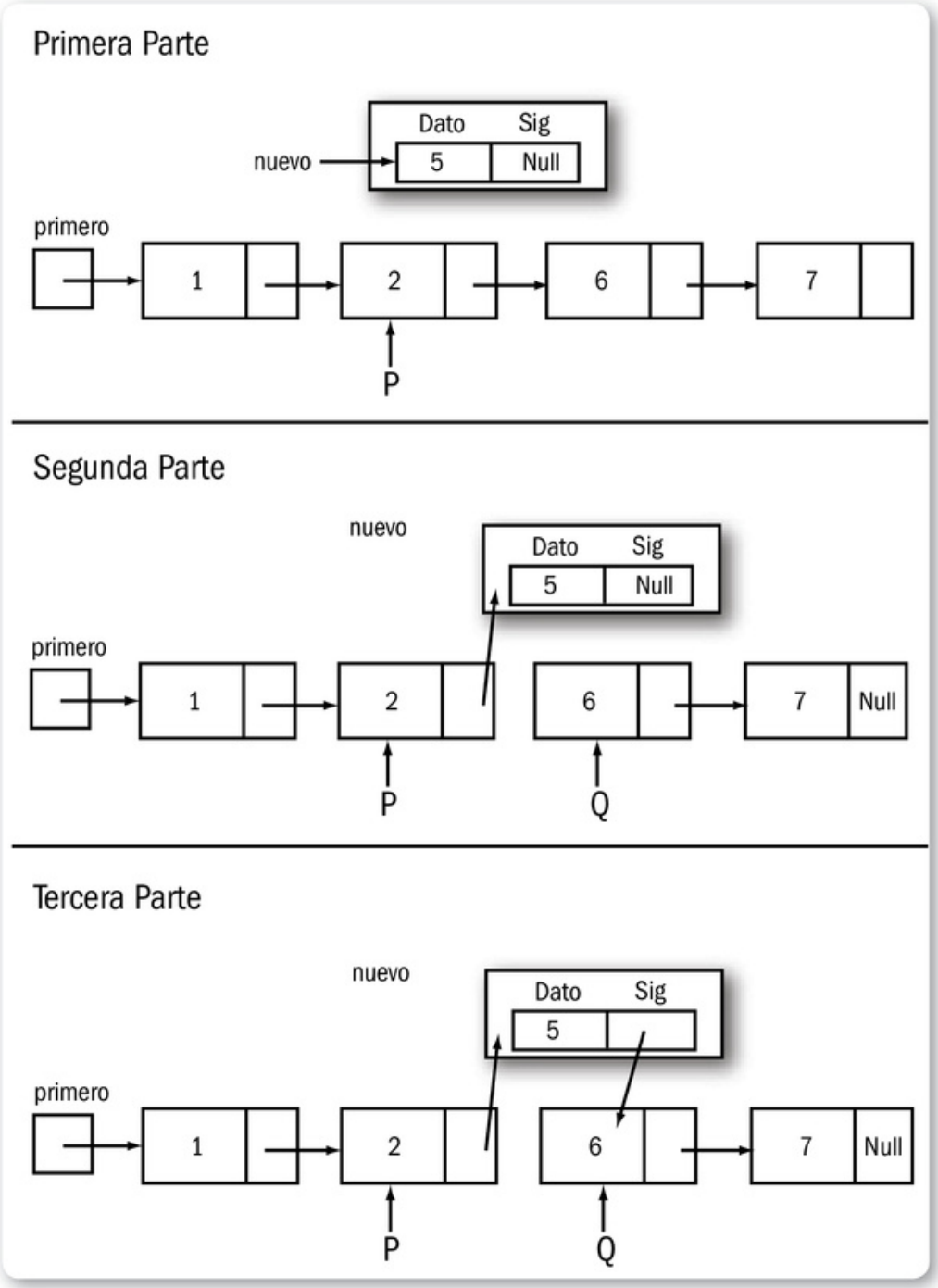


Figura 23. Representación en tres partes de cómo funciona la inserción entre nodos.

Por último, en caso de tener que insertar un nodo al final de la lista, podríamos utilizar un apuntador llamado **último**, que nos permitirá conocer cuál es el último nodo en ella.

PSEUDOCÓDIGO: Nodo mayor que el último

```
Inicio
        NUEVO<--Disponible
        Si NUEVO=NULO entonces

        Escribir "Desbordamiento de memoria"
Sino

        Si ultimo.dato < NvoInfo entonces

        Ultimo.sig=NUEVO

        NUEVO.sig<--NULL

        ultimo<--NUEVO
Fin si
Fin si
Fin
```

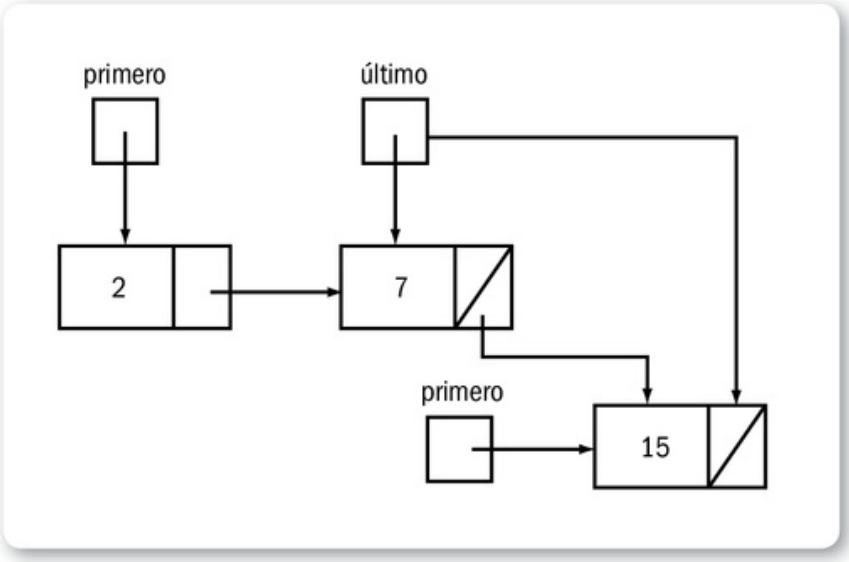


Figura 24. Representación de la inserción de un nodo después del último, y asignación de punteros.

Hasta aquí hemos visto diferentes formas de desenvolvernos en una lista enlazada a la hora de insertar nodos. Sin embargo, debemos tener en cuenta que si deseamos hacer una inserción en la lista, debemos preguntar por cada opción de las revisadas.

Veamos el código completo:

PSEUDOCÓDIGO: Insertar un nodo con valor desconocido

Variable primero, ultimo, P, Q, NUEVO **tipo nodo**

```
Inicio
        NUEVO<--Disponible
        Si NUEVO=NULO entonces

        Escribir "Desbordamiento de memoria"
        Sino
        Si NUEVO.dato>primero.dato entonces

        NUEVO.dato<--NvaInfo

        NUEVO.sig<--Primero
                        Primero<--NUEVO
                Sino

        Si NUEVO.dato<ultimo.dato entonces

        Ultimo.sig=NUEVO

        NUEVO.sig<--NULL
```

ALGORITMOS DE ORDENACIÓN

En la algoritmia se pueden encontrar diferentes técnicas que permiten realizar tareas de ordenación, de forma mucho más rápida y sencilla que otras técnicas. Algunas de ellas son: Insert-Sort, Shell-Sort, Merge-Sort y Quick-Sort. Entre ellas, se puede destacar a Quick-Sort, que es es actualmente el más eficiente y veloz de los métodos de ordenación de datos. Este también es conocido con el nombre de "método rápido" u "ordenamiento por partición".

```
        ultimo<--NUEVO
                        Sino
                                NUEVO.dato<--NvaInfo

        Q<--P.Sig

        P.Sig<--NUEVO

        NUEVO.sig<--Q
                        Fin si
                    Fin si
                Fin si
Fin
```

Eliminación de elementos

En el caso de la eliminación de nodos, debemos hacer que el nodo anterior a aquel que quiere eliminarse se enlace con el posterior a él, para que, así, el que queremos sacar quede fuera de la lista. Veamos cómo sería el algoritmo para esta operación:

PSEUDOCÓDIGO: Eliminar un nodo

```
QßP.Sig //En la figura el paso 2
P.SigßQ.Sig //En la figura el paso 3
LiberarNodo(Q) //En la figura el paso 4
```

EL ESTUDIO DE LAS ESTRUCTURAS

El estudio de las estructuras de datos nos permitirá acceder a una parte fundamental dentro de nuestros desarrollos, que es el DER. Se trata de un Diagrama de Entidad Relación que se utiliza para realizar el correcto diseño de las estructuras de una base de datos, en la cual luego desarrollaremos las consultas y registros de información. En un DER, cada entidad se representa mediante un rectángulo, cada relación mediante un rombo y cada dominio mediante un círculo.

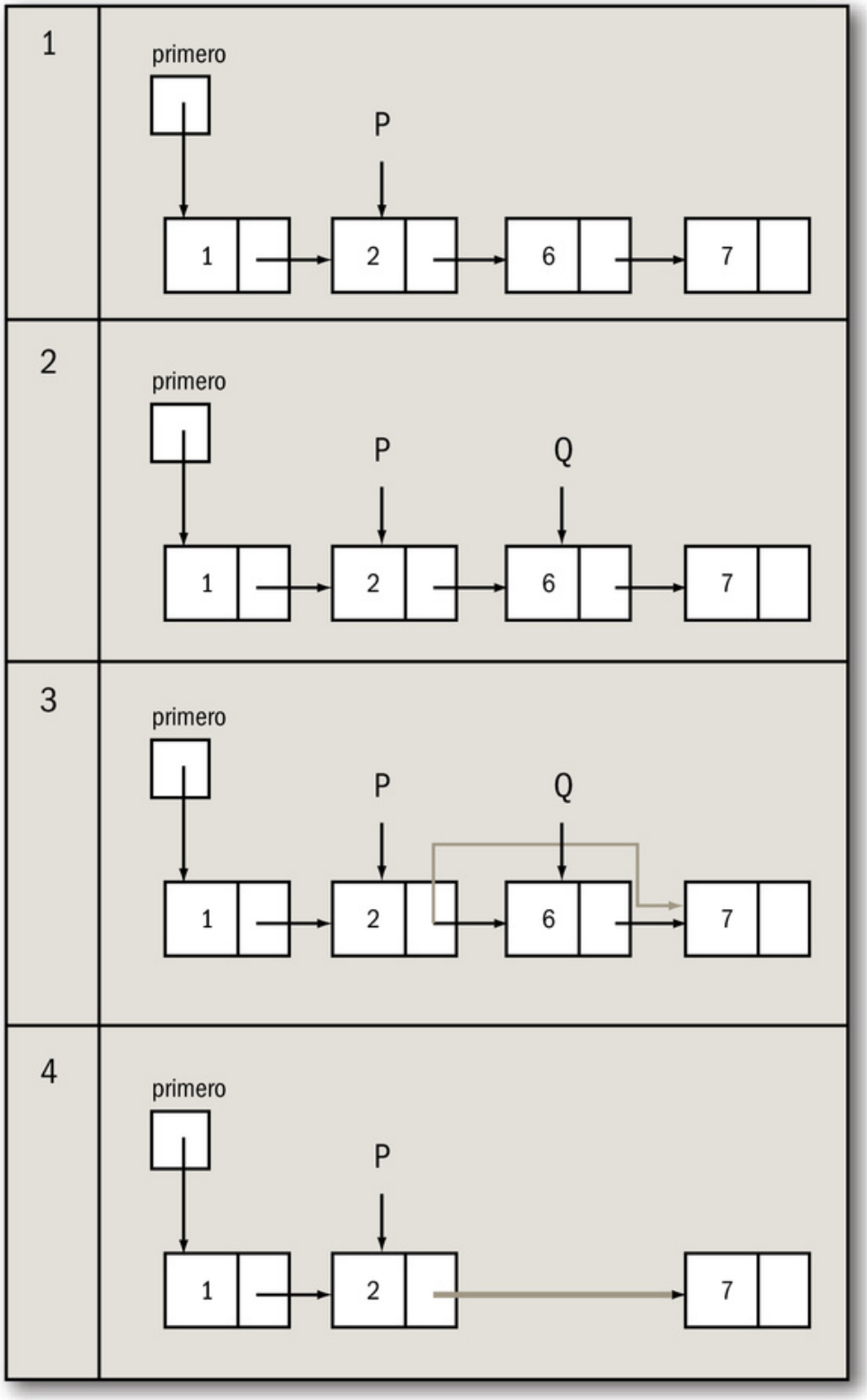


Figura 25. Representación de cómo se elimina un nodo y qué sucede con los punteros.

Tengamos en cuenta que el proceso de borrado nos dará un lugar para nodos libres o disponibles. Dicho espacio de memoria puede ser reutilizado si codificamos aquel apuntador llamado **Disponible**.

A continuación, veamos cuáles son los distintos escenarios posibles después de la eliminación:

- Eliminar un único nodo.
- Eliminar el primer nodo.
- Eliminar el último nodo.
- Eliminar un nodo en medio de otros.

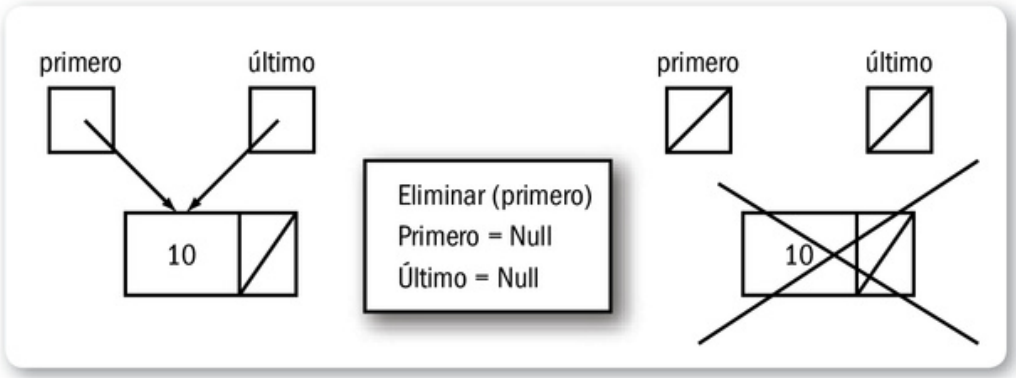


Figura 26. Esta imagen nos muestra lo que sucede si eliminamos un nodo único.

PSEUDOCÓDIGO: Eliminar el primero nodo

```
Inicio
        Si primero=NULO entonces

        Escribir "No hay elementos para eliminar"
Sino

//Si existe sólo un elemento y es el que se debe eliminar

        Si (primero.sig=NULO) y (NvaInfo=primero.dato) entonces


        Eliminar(primero) //libera memoria, elimina el nodo
```

```
                    Primero=NULO
                    Ultimo=NULO
                Sino
//Si existe más de un elemento

Si (NvaInfo=primero.dato) entonces
                    P=primero //guarda la dirección del primer nodo.
                    Primero=P.Sig //guarda la dirección del segundo nodo.
                    Eliminar(P) //libera memoria, elimina el nodo
                    Sino
                    .../continua con otras instrucciones
                    Fin si
                Fin si
            Fin si
Fin
```

A continuación, veremos destacado el siguiente código que se utiliza para quitar el primer nodo de una lista:

```
P=primero
Primero=P.Sig
Eliminar(P)
```

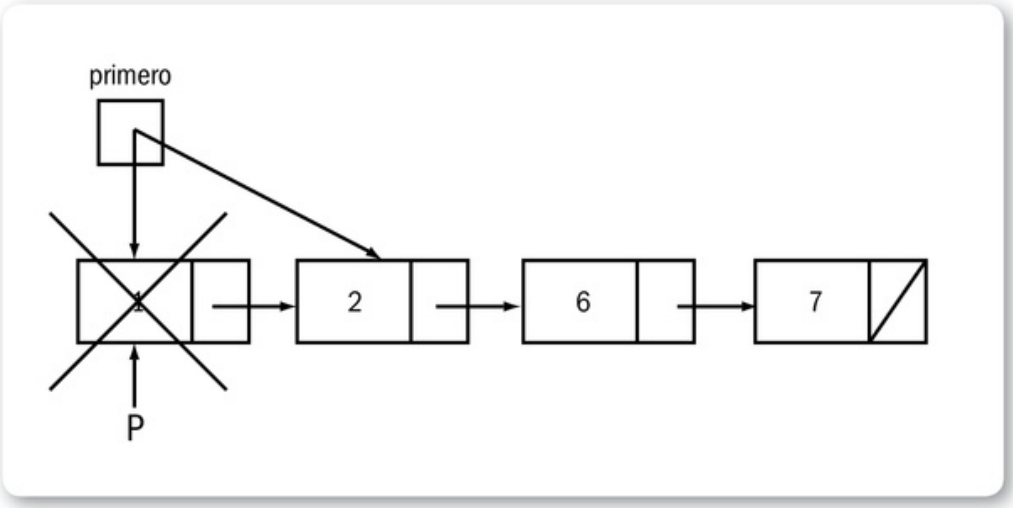


Figura 27. Esta imagen nos muestra lo que sucede si eliminamos el primer nodo.

PSEUDOCÓDIGO: Eliminar el último nodo

```
                Inicio
                    Si primero=NULO entonces

Escribir "No hay elementos para eliminar"
                Sino
                //Si existe sólo un elemento y es el que se debe eliminar

Si (primero.sig=NULO) y (NvaInfo=primero.dato) entonces

Eliminar(primero) //libera memoria, elimina el nodo
                Primero=NULO
                Ultimo=NULO
                Sino
                //Si existe más de un elemento

Si (NvaInfo=primero.dato) entonces
                    P=primero
                    Primero=P.Sig
                    Eliminar(P)
                Sino
                    P=primero
                    Q=primero
                    Mientras (NvaInfo<>P.dato) y (P<>NULO)
                     Q=P
                     P=P.Sig
                    Fin Mientras

                    Si (P<>NULO) //si encontró el valor...
                     Si (NvaInfo=ultimo.dato)
                       Q.Sig=NULO
                       ultimo=Q
                       eliminar(ultimo)
                     Fin si
                    Fin si
```

```
                    ...//continua con otras instrucciones
                Fin si

                Fin si
                Fin si
                Fin
```

A continuación, veremos destacado el siguiente código que se utiliza para quitar el último nodo:

```
Q.Sig=NULO
ultimo=Q
eliminar(ultimo)
```

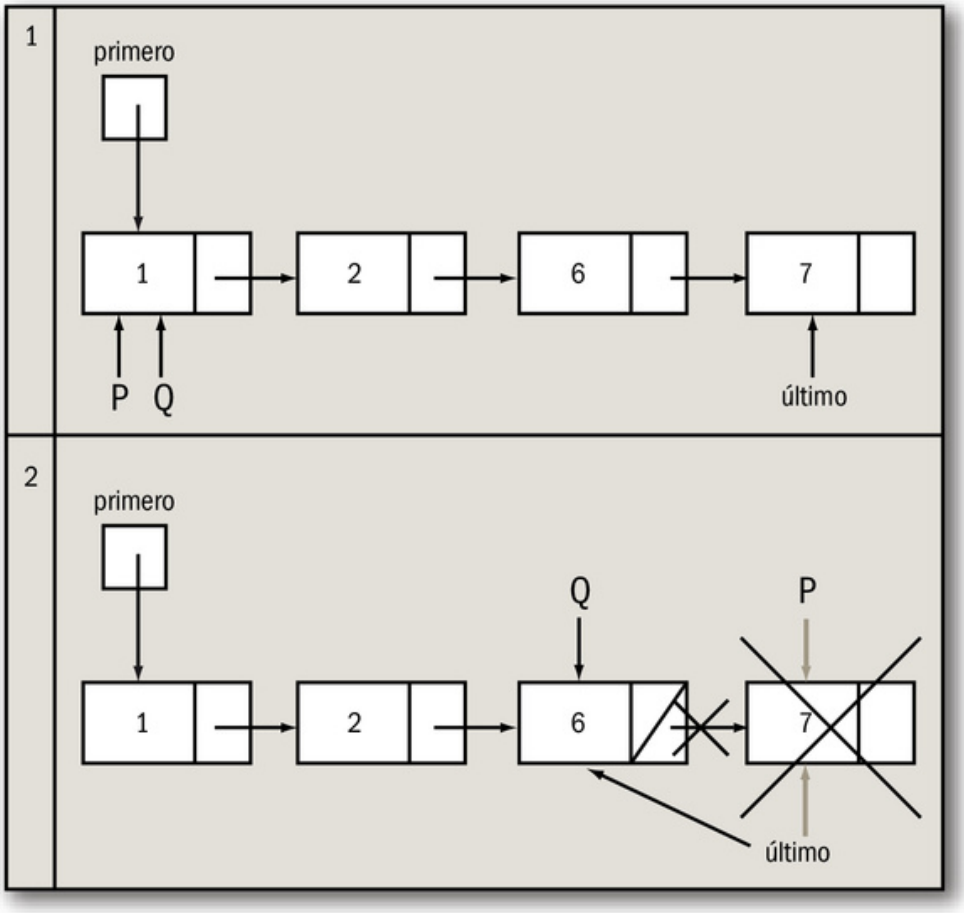


Figura 28. Esta imagen nos muestra lo que sucede si eliminamos el último nodo.

PSEUDOCÓDIGO: Eliminar un nodo entre otros

```
                Inicio
                    Si primero=NULO entonces

Escribir "No hay elementos para eliminar"
                Sino
                //Si existe sólo un elemento y es el que se debe eliminar

Si (primero.sig=NULO) y (NvaInfo=primero.dato) entonces

Eliminar(primero) //libera memoria, elimina el nodo
                Primero=NULO
                Ultimo=NULO
                Sino
                //Si existe más de un elemento

Si (NvaInfo=primero.dato) entonces
                    P=primero
                    Primero=P.Sig
                    Eliminar(P)
                Sino
                    P=primero
                    Q=primero
                    Mientras (NvaInfo<>P.dato) y (P<>NULO)
                     Q=P
                     P=P.Sig
                    Fin Mientras

                    Si (P<>NULO) //si encontró el valor...
                     Si (NvaInfo=ultimo.dato)
                       Q.Sig=NULO
                       ultimo=Q
                       eliminar(ultimo)
                     sino
                       Q.sig=P.sig
                       Eliminar(P)
```

```
Fin si
    Fin si
Fin si

Fin si
Fin si
Fin
```

Destacamos el código que se utiliza para quitar un nodo de entre medio de otros; también lo veremos en la **Figura 29**:

Q.sig=P.sig

Eliminar(P)

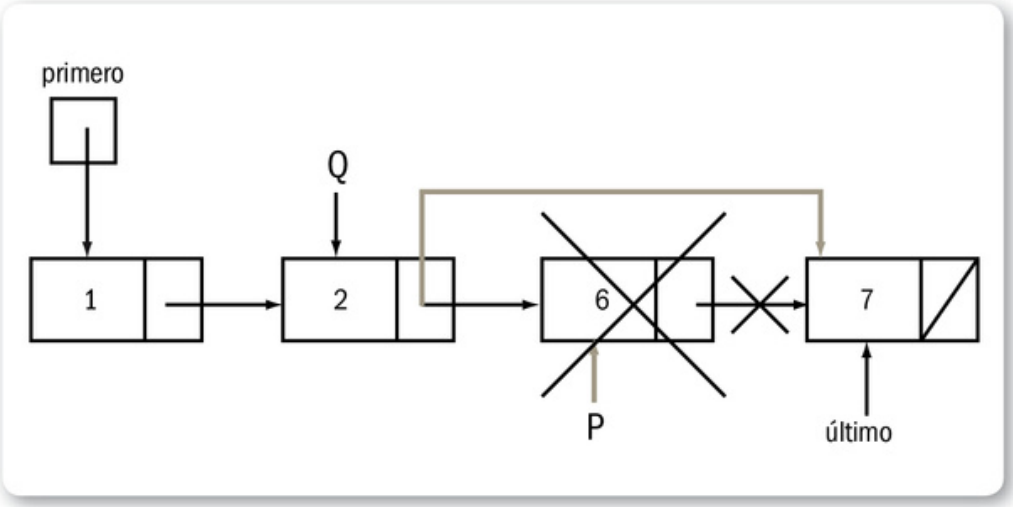


Figura 29. Esta imagen nos muestra lo que sucede si eliminamos un nodo entre otros.

Como podemos observar, debemos enlazar el nodo anterior con el posterior al dato que vamos a eliminar. En este caso, eliminamos el nodo 6 y enlazamos el nodo 2 con el 7.

Hasta aquí hemos visualizado y ejercitado las diferentes operaciones que podemos realizar sobre las estructuras de datos **lista enlazada**: **insertar**, **eliminar**, **buscar** y **recorrer**.

A continuación, vamos a desarrollar otro tipo de estructura, que está estrechamente relacionada con la lista.

Listas doblemente enlazadas

Esta estructura deriva directamente del tipo de lista y es lineal, ya que cada nodo tiene dos enlaces: uno al nodo siguiente, y otro al anterior.

Las listas doblemente enlazadas no necesitan un nodo especial para acceder a ellas, sino que pueden recorrerse en ambos sentidos a partir de cualquier nodo hasta llegar a uno de los extremos.

PSEUDOCÓDIGO	C++
Estructura Nodo	struct nodo
Variable dato tipo entero	{
Estructura nodo siguiente	int dato;
Estructura nodo anterior	struct nodo *siguiente;
Fin Estructura	struct nodo *anterior;
	}
Variable primero tipo nodo =NULO	
Variable último tipo nodo =NULO	
Variable Lista tipo nodo =NULO	

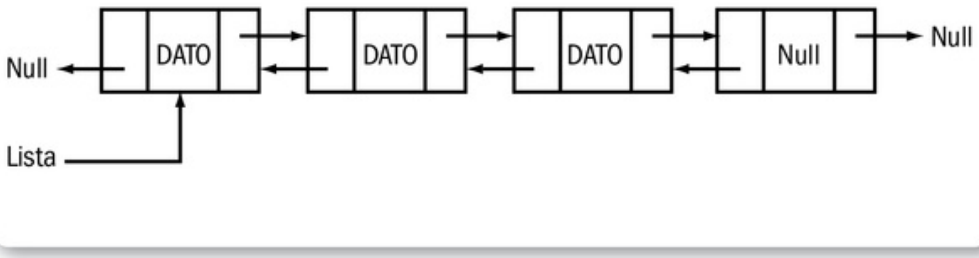


Figura 30. Representación de una lista doblemente enlazada, donde podemos observar cómo funcionan los punteros.

A continuación, vamos a revisar las mismas acciones que hicimos con las listas enlazadas simples. En este caso, precisaremos dos variables auxiliares: una que almacene la posición del primer nodo, y otra que almacene la posición del último.

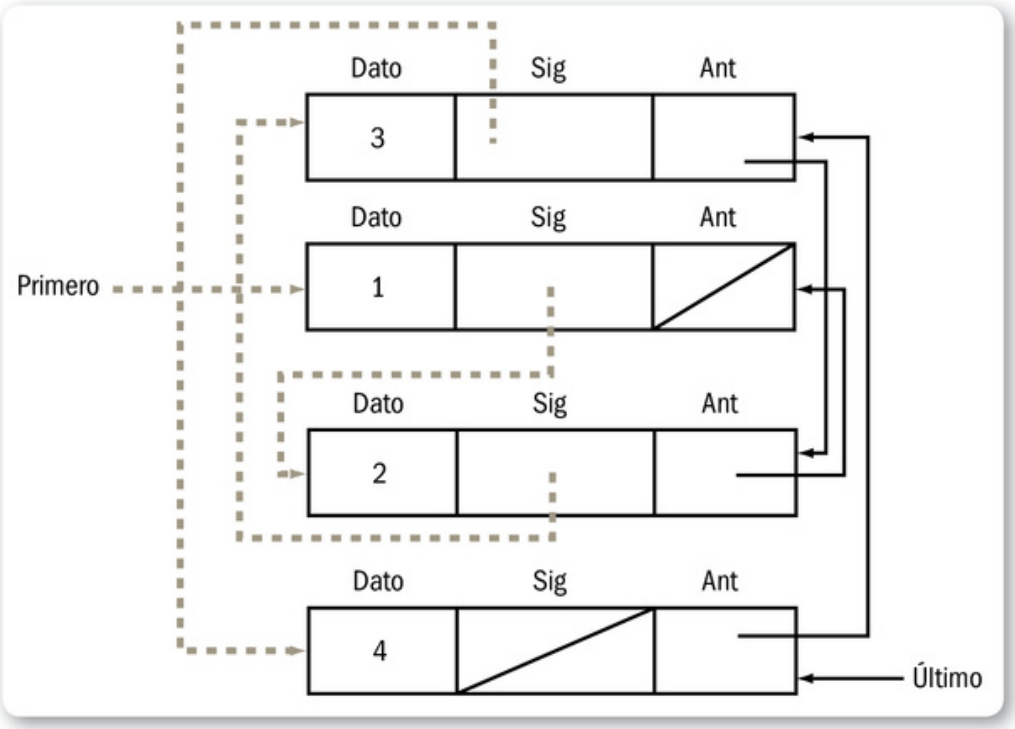


Figura 31. En la imagen vemos un ejemplo de lista doblemente enlazada, y las relaciones que pueden llegar a tener los punteros.

Creación

Así como antes vimos la creación de estructuras de listas enlazadas, ahora veremos los diferentes escenarios que pueden presentarse en el caso de las doblemente enlazadas.

PSEUDOCÓDIGO

Algoritmo Crear Lista Doble

```
Inicio
    Variable Nuevo, Primero, ultimo tipo nodo
    Variable AuxDato tipo numérico

    Si Primero=nulo // Y para almacenar el dato se controla si no existen
       elementos.
        Nuevo = Disponible
```

```
            Si Nuevo <> nulo // Se controla que haya memoria, si no
                devolverá Null.
                  Leer AuxDato //traerá la información a almacenar

                  Nuevo.dato = AuxDato
                  Nuevo.sig = NULO
                  Nuevo.ant = NULO

                  Primero = Nuevo
                  Ultimo = Nuevo
               Si no
                  Escribir "No hay memoria suficiente"
               Fin Si
            Si no
               Escribir "La lista ya está creada"
            Fin Si
Fin
```

Inserción de elementos

La inserción se debe hacer a la izquierda del nodo apuntado por la posición ofrecida a la función insertar. Esto implica que al insertar un nodo, el puntero utilizado sigue apuntando al mismo elemento. A continuación, veamos los diferentes escenarios que podemos encontrar dentro de una lista doble.

- Si la lista está vacía: En este caso, supongamos que queremos insertar un nodo, cuyo puntero que define la lista vale nulo.

LISTA CIRCULAR

Una lista circular es una lista lineal en la que el último nodo apunta al primero. Las listas circulares evitan excepciones en las operaciones que se realicen sobre ellas. Cada nodo siempre tiene uno anterior y uno siguiente. En algunas se añade un nodo especial de cabecera, de modo que se evita la única excepción posible: que la lista esté vacía.

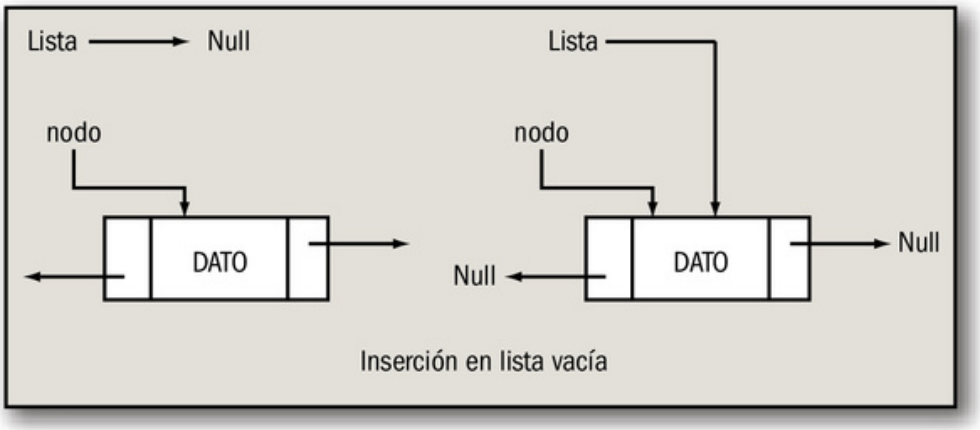


Figura 32. Representación de la lista doblemente enlazada, pero estando vacía.

Observamos el ejemplo en la **Figura 32**. Debemos considerar que:

1. **Lista** apunta a nodo.
2. **Lista.siguiente** y **lista.anterior** son igual a **nulo**.
   //Lista sería una variable de tipo puntero.

- Si debemos insertar un nodo en la primera posición de la lista: Tomaremos una lista que no esté vacía. Consideraremos que **lista** apunta al primer nodo, veamos la **Figura 33**.

Para este ejemplo, usaremos una lista que no está vacía. En la **Figura 33** notaremos lo que sucede en una lista doblemente enlazada, en donde la Lista apunta al primer nodo.

ESTRUCTURA DE ÁRBOL

Un árbol es una estructura no lineal en la que cada nodo puede apuntar a uno o varios nodos. También se suele dar una definición recursiva: un árbol es una estructura compuesta por un dato y varios árboles. Los árboles tienen otra característica importante: cada nodo solo puede ser apuntado por otro nodo, es decir, cada nodo tendrá un único padre. Esto hace que estos árboles estén fuertemente jerarquizados, y es lo que en realidad les da la apariencia de árboles.

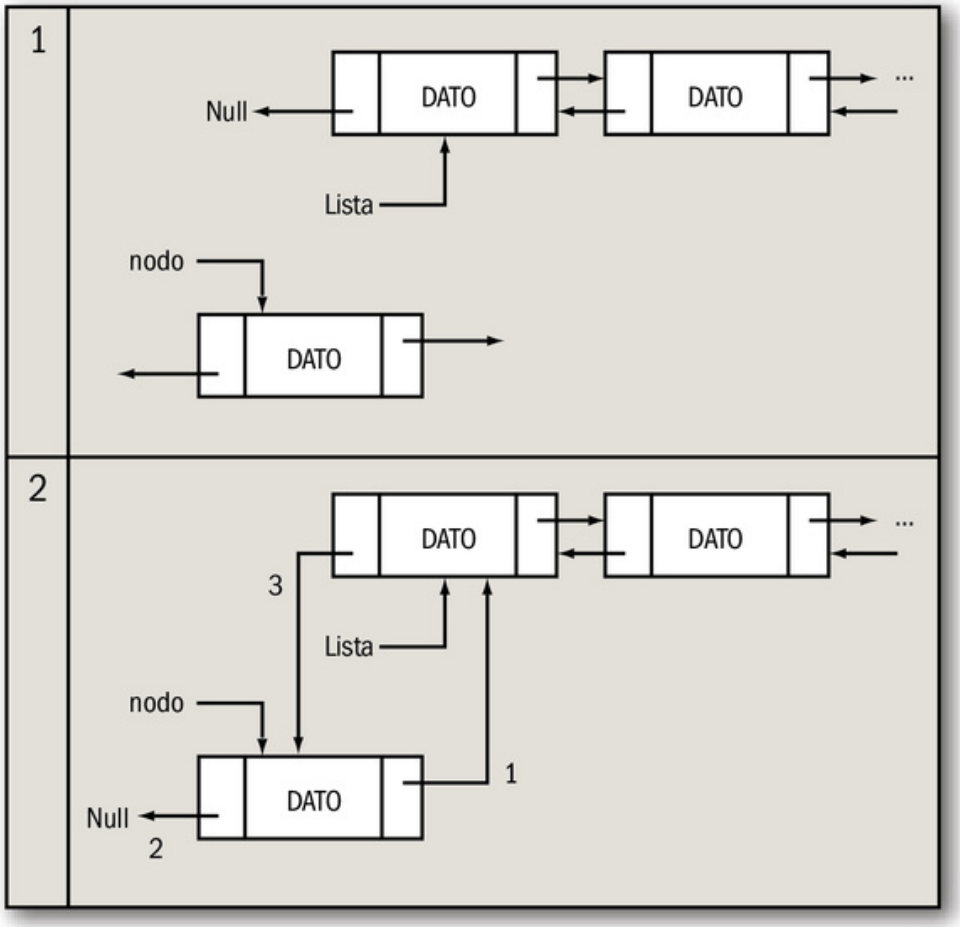


Figura 33. Representación de lo que sucede en una lista doblemente enlazada al ejecutar los pasos vistos antes.

Observemos la lista que se detalla en el punto 2 de la figura y leamos los pasos siguientes:

1. El puntero **nodo.siguiente** debe apuntar a **Lista**.
2. El puntero **nodo.anterior** apuntará a **Lista.anterior**.
3. **Lista.anterior** debe apuntar a **nodo**.

Recordemos que el puntero **Lista** no necesariamente apunta a un miembro concreto de la lista doble, ya que cualquier elemento apuntador es válido como referencia.

- Si debemos insertar un nodo en la última posición de la lista: Iniciaremos el ejemplo con el apuntador **Lista** haciendo referencia al último elemento que se encuentra en la lista doble.

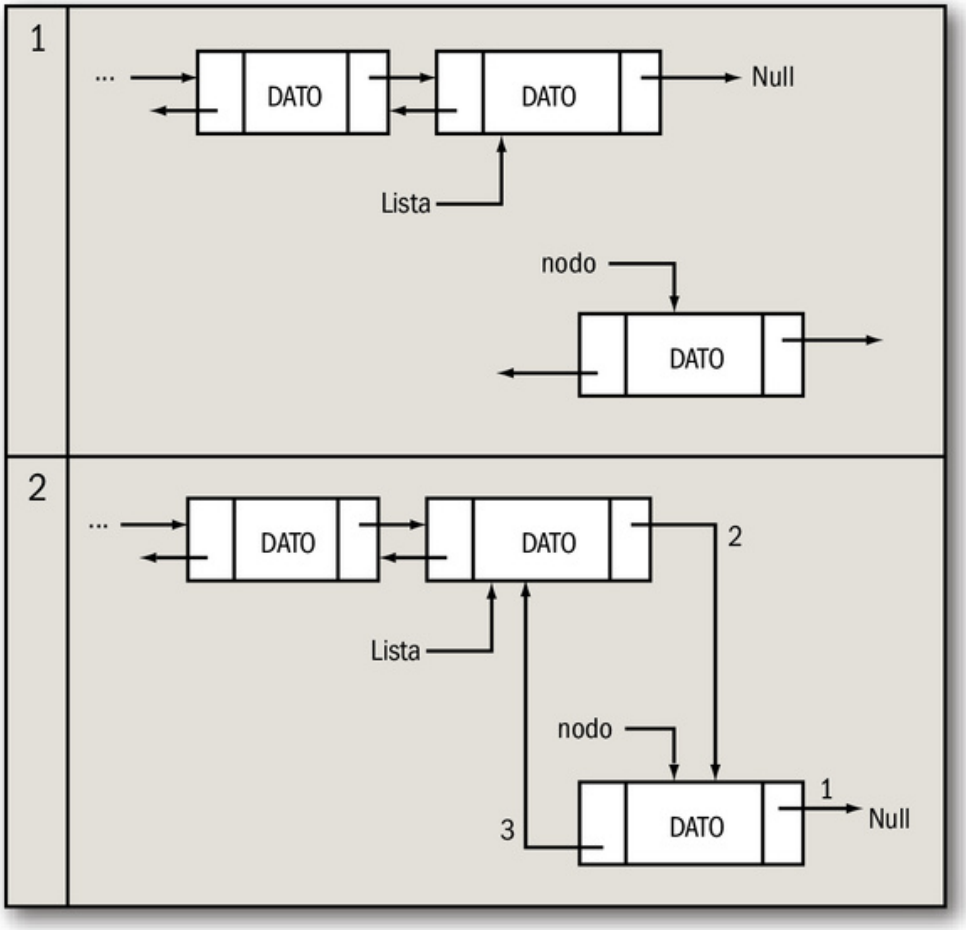


Figura 34. Representación de lo que sucede en una lista doblemente enlazada al insertar un nodo en la última posición.

Observando la lista del punto 2, leamos los siguientes pasos:

1. **nodo.siguiente** debe apuntar a **Lista.siguiente**, que tiene el valor nulo.
2. **Lista.siguiente** debe apuntar a **nodo**.
3. **nodo.anterior** apuntará a **Lista**.

OTRAS ESTRUCTURAS ÁRBOL

Existen otras estructuras que se utilizan para distintas tareas, como Arboles AVL, con estructuras más optimizadas que permiten reducir los tiempos de búsqueda, y Arboles B, sostenidos por estructuras más complejas que optimizan aún más los resultados.

- Si debemos insertar un nodo entre otros nodos de la lista: En caso de tener que insertar un nodo en cualquier parte de una lista ya constituida, lo que debemos realizar es lo que nos indica la representación gráfica que aparece en la **Figura 35**.

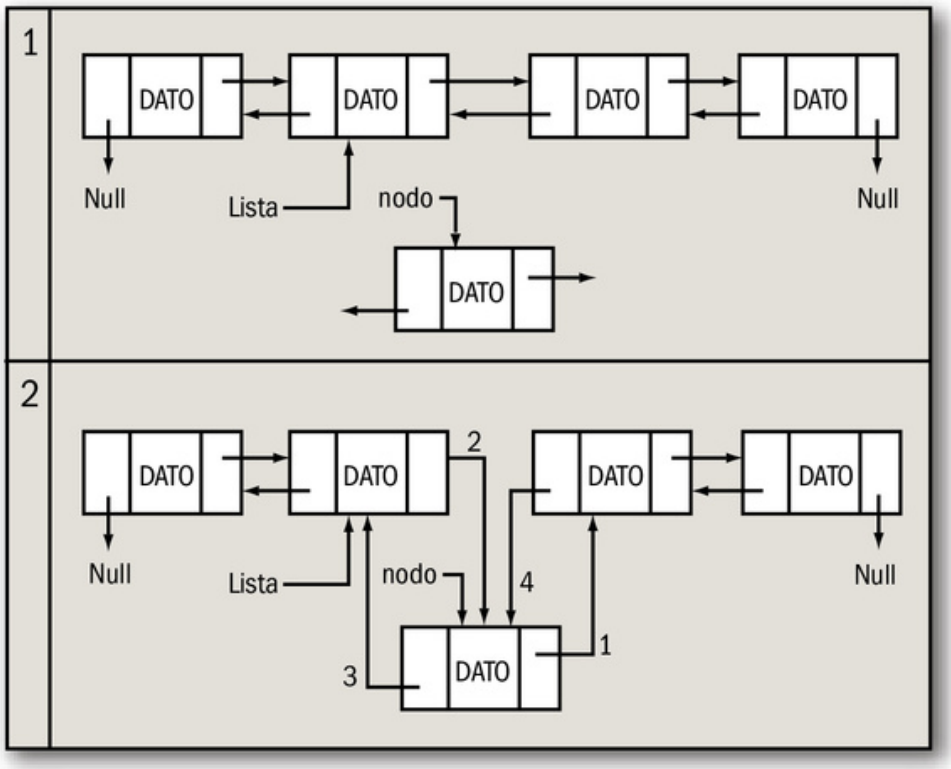


Figura 35. Representación de lo que sucede en una lista doblemente enlazada al insertar un nodo entre otros nodos de la lista.

Observemos la lista que se detalla en el punto 2 de la figura y leamos los pasos siguientes:

1. Primero **nodo.siguiente** apunta a **lista.siguiente**.
2. Luego **Lista.siguiente** apunta a **nodo**.
3. **nodo.anterior** apunta a **lista**.
4. Y por último, **nodo.siguiente.anterior** apunta a **nodo**.

Aquí trabajamos como si tuviéramos dos listas enlazadas: primero insertamos elementos en una lista abierta; y luego, realizamos lo mismo con la lista que enlaza los nodos en sentido contrario.

En el último paso tenemos un puntero auxiliar llamado **p** que, antes de empezar a insertar, apunta al nodo que continúa en la lista de elementos. La sintaxis para hacer esta indicación es **p = Lista.siguiente**.

Cerrando un poco el tema de las listas doblemente enlazadas, al momento de programarlas debemos tener en cuenta lo siguiente:

INSERCIÓN	
▼ DESCRIPCIÓN DE PASOS	▼ UTILIDAD
Si la lista está vacía, hacemos que Lista apunte a nodo. Y nodo.anterior y nodo.siguiente, a NULO.	Lista vacía
Si la lista no está vacía, hacemos que nodo.siguiente apunte a Lista.siguiente	Insertar dentro de una lista en un sentido
Después que Lista.siguiente apunte a nodo	Insertar dentro de una lista en un sentido
Hacemos que nodo.anterior apunte a Lista	Insertar dentro de una lista en sentido contrario
Si nodo.siguiente no es NULO, entonces hacemos que nodo.siguiente.anterior apunte a nodo	Insertar dentro de una lista en sentido contrario

Tabla 2. Inserción en listas dobles.

Para completar la inserción de elementos en listas dobles enlazadas, veamos cómo sería la codificación del pseudocódigo para las diferentes situaciones que vimos anteriormente:

```
Algoritmo Insertar
        Variable Nuevo, Primero, Ultimo tipo nodo
        Variable P tipo nodo //puntero auxiliar para recorrer
    Variable AuxDato tipo numérico entero
```

```
Inicio
Si Primero <> nulo
        Nuevo = Disponible
```

```
        Si Nuevo <> nulo
              Leer AuxDato
              Nuevo.Dato = AuxDato

              Si AuxDato < primero.Dato
                    Nuevo.ant = nulo
                    Nuevo.sig = primero
                    primero.ant = nuevo
                    primero = nuevo
              Si no
                 Si AuxDato > ultimo.Dato
                    ultimo.sig = Nuevo
                    Nuevo.sig = null del nuevo dato, null
                    ultimo = Nuevo
                    Nuevo.ant = ultimo
                 Si no
                    P = primero

                    Mientras AuxDato > P.dato
                    P = P.sig
                    Fin mientras

                    Nuevo.ant = P.ant
                    Nuevo.sig = P
                    P.ant.sig = nuevo
                    P.ant = nuevo
              Fin Si
           Fin Si

     Si no
           Escribir "No existe memoria"
     Fin Si

Si no
        Escribir "La lista no existe"

Fin si
```

Recorrido

Podemos recorrer la información de la lista doble en dos sentidos: ascendente, comenzando por el primer nodo; o descendente, empezando en el último. El algoritmo es similar al de las listas simples.

PSEUDOCÓDIGO

Algoritmo Recorrer ascendente
    Variable P tipo nodo

```
Inicio
    P = primero
        Mientras (P <> nulo)
                Escribir P.dato
                P = P.sig
        Fin mientras
Fin
```

PSEUDOCÓDIGO

Algoritmo Recorrer descendente
    Variable P tipo nodo

```
Inicio
    P = ultimo
        Mientras (P <> nulo)
                Escribir P.dato
                P = P.ant
        Fin mientras
Fin
```

HASH

La tabla hash es una estructura de datos que asocia llaves o claves con valores. La operación principal que soporta de manera eficiente es la búsqueda: permite el acceso a los elementos (por ejemplo, teléfono y dirección) a partir de una clave generada (por ejemplo, usando el nombre o número de cuenta).

Eliminación de elementos

A continuación, veamos cuáles son los diferentes escenarios posibles en el manejo de una lista doble enlazada.

- Eliminar el único nodo en una lista doblemente enlazada: En este caso, el **nodo** será apuntado por **Lista**. El proceso es simple:

1. Eliminamos el **nodo**.
2. Hacemos que **Lista** apunte a **NULO**.

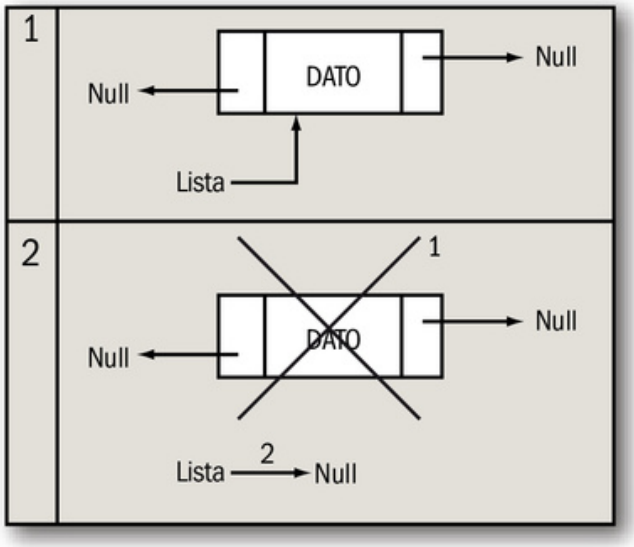


Figura 36. Imagen que muestra cómo se elimina el único nodo en una lista doblemente enlazada.

- Eliminar el primer **nodo** de una lista doblemente enlazada: Puede suceder que el nodo por borrar esté apuntado por **Lista** o no. Si lo está, simplemente hacemos que **Lista** sea **Lista.siguiente**.

LAS ESTRUCTURAS Y C / C++

Las estructuras básicas disponibles en C y C++ (structs y arrays) tienen una importante limitación: no pueden cambiar de tamaño durante la ejecución. Los arrays están compuestos por un determinado número de elementos que se decide en la fase de diseño, antes de que el programa ejecutable sea creado.

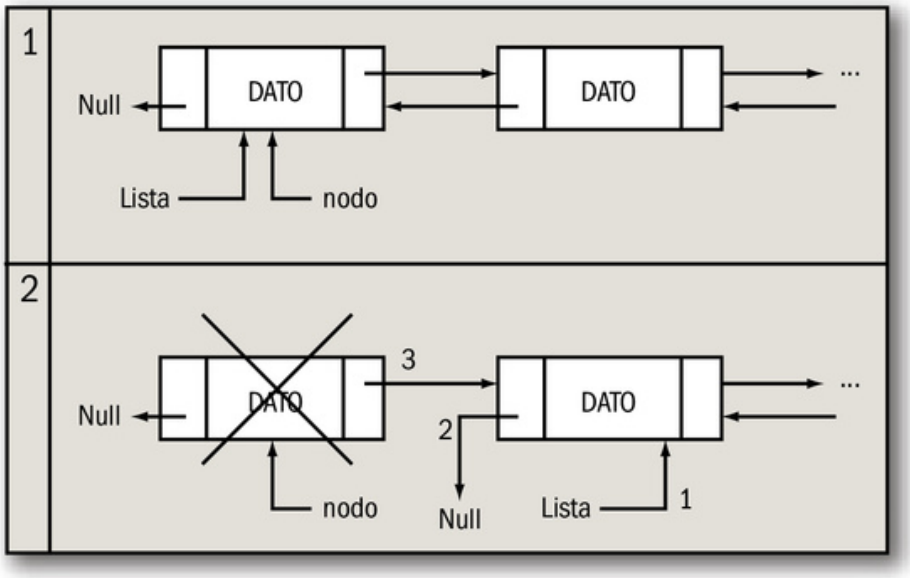


Figura 37. Representación de lo que sucede al eliminar el primer nodo de una lista doblemente enlazada.

1. Si **nodo** apunta a **Lista**, hacemos que **Lista** apunte a **Lista.siguiente**.
2. Hacemos que **nodo.siguiente.anterior** apunte a **NULO**.
3. Borramos el nodo apuntado por **nodo**.

El paso 2 separa el nodo a borrar del resto de la lista, independientemente del nodo al que apunte Lista.

Veamos a continuación un pseudocódigo que nos permita borrar el primer nodo.

PSEUDOCÓDIGO

Algoritmo Eliminar lista doble
Var auxDato **tipo** numérica
Var P **tipo** ejemplo

```
Inicio
        Si Primero=null
                Escribir "No hay elementos para eliminar"
        Si no
                Leer AuxDato
```

```
Si (primero.sig = nulo) y (auxDato = primero.dato)
                    Eliminar (primero)
                    primero = nulo
                    ultimo = nulo
                Si no
                    Si (auxDato = primero.dato)
                      P= primero
                      primero = primero.sig
                      primero.ant = NULO
                      LiberarMemoria(P)
                Si no
                ...//otras instrucciones
                Fin si
            ...//otras instrucciones
            Fin si
Fin
```

- Eliminar el último nodo de una lista doblemente enlazada: Nuevamente tenemos los dos casos posibles: que el nodo por borrar esté apuntado por **Lista** o que no lo esté. Si lo está, simplemente hacemos que **Lista** sea **Lista.anterior**.

1. Si nodo apunta a **Lista**, hacemos que **Lista** apunte a **Lista.anterior**.
2. Hacemos que **nodo.anterior.siguiente** apunte a **NULO**.
3. Borramos el **nodo** apuntado por **nodo**.

El paso 2 depara el nodo a borrar del resto de la lista, independientemente del nodo al que apunte **Lista**.

LISTA CIRCULAR SIMPLE

Cada nodo tiene un enlace similar al de las listas enlazadas simples, excepto que el siguiente nodo del último apunta al primero. Esto nos permite rápidas inserciones al principio y accesos al primer nodo desde el puntero del último nodo.

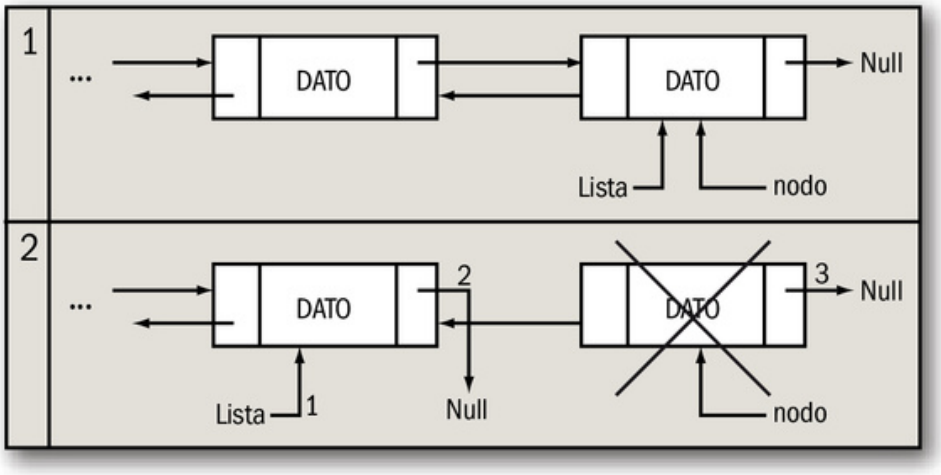


Figura 38. Imagen que muestra lo que sucede al eliminar el último nodo de una lista doblemente enlazada.

PSEUDOCÓDIGO

Algoritmo Eliminar lista doble
Var auxDato **tipo** numérica
Var P **tipo** ejemplo

```
Inicio
        Si Primero=null
                Escribir "No hay elementos para eliminar"
        Si no
                Leer AuxDato
                Si (primero.sig = nulo) y (auxDato = primero.dato)
                        Eliminar (primero)
```

SCHEME

Es un lenguaje funcional y un dialecto de Lisp. Fue desarrollado por Guy L. Steele y Gerald Jay Sussman en la década del 70 e introducido en el mundo académico a través de una serie de artículos conocidos como los Lambda Papers de Sussman y Steele.

```
primero = nulo
                        ultimo = nulo
                Si no
                        Si (auxDato = primero.dato)
                                P= primero
                                primero = primero.sig
                                primero.ant = NULO
                                LiberarMemoria(P)
                          Si no
                                Si (auxDato = ultimo.dato)
                                    ultimo = ultimo.ant
                                    ultimo.sig = NULO
                                    LiberarMemoria(P)
                                Si no
                                ...//otras instrucciones
                                Fin si
                        ...//otras instrucciones
                        Fin si
                ...//otras instrucciones
                Fin si
        ...//otras instrucciones
        Fin si
Fin
```

- Eliminar un nodo intermedio de una lista doblemente enlazada: Una vez más, nos enfrentamos con dos casos posibles: que el nodo por borrar esté apuntado por **Lista**, o que no lo esté. Si lo está, simplemente hacemos que **Lista** sea **Lista.anterior** o **Lista.siguiente**. Se trata de un caso más general que los dos anteriores.

NODO CENTINELA

A veces, las listas enlazadas tienen un nodo centinela (también llamado falso nodo o nodo ficticio) al principio o al final, el cual no es usado para guardar datos. Su propósito es simplificar o agilizar las operaciones, asegurando que cualquier nodo tenga otro anterior o posterior.

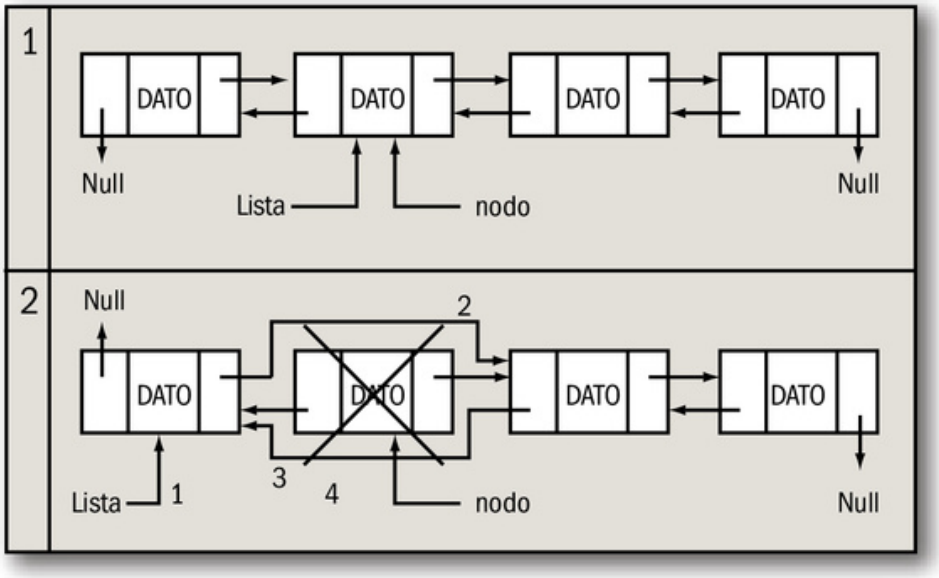


Figura 39. Representación de lo que sucede al eliminar un nodo intermedio de una lista doblemente enlazada.

1. Si nodo apunta a **Lista**, hacemos que **Lista** apunte a **Lista.anterior** (o **Lista.siguiente**).
2. Hacemos que **nodo.anterior.siguiente** apunte a **nodo.siguiente**.
3. Hacemos que **nodo.siguiente.anterior** apunte a **nodo.anterior**.
4. Borramos el nodo apuntado por **nodo**.

Cuando eliminamos un nodo intermedio, **primero** y **último** no se modifican; pero sí cambiarán los punteros **sig** del nodo menor y el **ant.** del nodo mayor. Es bueno recordar que podemos utilizar la variable auxiliar **P** para recorrer la lista, la cual apuntará al dato por eliminar.

LISTAS DOBLES VS. SIMPLES

Las listas doblemente enlazadas requieren más espacio por nodo, y sus operaciones básicas resultan más costosas pero ofrecen una mayor facilidad para manipular, ya que permiten el acceso secuencial a la lista en ambas direcciones. Es posible insertar o borrar un nodo en un número fijo de operaciones dando únicamente la dirección de dicho nodo.

PSEUDOCÓDIGO

Algoritmo Eliminar lista doble
Var auxDato **tipo** numérica
Var P **tipo** ejemplo

```
Inicio
        Si Primero=null
                Escribir "No hay elementos para eliminar"
                 Si no
                        Leer AuxDato
                        Si (primero.sig = nulo) y (auxDato = primero.dato)
                            Eliminar (primero)
                            primero = nulo
                            ultimo = nulo
                        Si no
                            Si (auxDato = primero.dato)
                                    P= primero
                                    primero = primero.sig
                                    primero.ant = NULO
                                    LiberarMemoria(P)
                             Si no
                                  Si (auxDato = ultimo.dato)
                                        ultimo = ultimo.ant
                                        ultimo.sig = NULO
                                        LiberarMemoria(P)
                                  Si no
                                          P = primero
                                          Mientras (AuxDato <> P.dato) y (P<> null)
                                          P = P.sig
                                          Fin mientras

                                          Si P <> nulo
                                             P.sig.ant = P.ant
                                             LiberarMemoria(P) // Elimina el nodo.
                                          Si no
                                        Escribir "El dato a eliminar no se encuentra"
                                        Fin Si
```

```
                                    Fin si
                                Fin si
                            Fin si
                        Fin si
Fin
```

Concluyendo con las estructuras de datos que hemos estudiado a lo largo del capítulo, podemos hacer hincapié en el manejo de memoria dinámica con listas enlazadas (simples y dinámicas).

Ahora veamos qué aplicaciones podemos dar a las listas enlazadas. Por ejemplo, antes mencionamos que suelen utilizarse para el desarrollo en el área de inteligencia artificial. Sin irnos tan lejos, podemos observar que los actuales juegos de ocio que encontramos en Internet suelen basarse en alguna estructura muy similar.

Veamos un caso en la **Figura 40**.

Figura 40. Juego que utiliza una lista de acciones que podemos considerar una cola de objetos o valores.

Como vemos en la **Figura 40**, el juego presentado se llama PALADOG (**http://armorgames.com/play/13262/paladog**) y nos muestra el funcionamiento de una lista enlazada. Si nos ubicamos como jugadores, notaremos que las acciones se van cargando en la lista y, si hacemos clic sobre ellas, se irán "eliminando", al mismo tiempo que las restantes se van reordenando.

Tambien podemos pensarlo desde un ejemplo más sencillo, como puede ser el juego de ajedrez. En caso de que el contrincante sea una computadora, esta contará con muchos movimientos posibles en una lista, que irá reduciendo en función de nuestras jugadas. Si analizamos las ventajas que puede tener la computadora, debemos considerar que: existen 20 primeros movimientos posibles para las blancas, y otros tantos para las negras; por lo tanto, se pueden formar **400 posiciones** distintas tras la primera jugada de cada bando. Para el segundo movimiento de las blancas, la situación se complica: hay **5.362 posiciones** posibles cuando las blancas hacen su segunda jugada.

> LA ESTRUCTURA DINÁMICA SE CONVIERTE EN LA INTELIGENCIA DE LA COMPUTADORA

De esta forma, podemos visualizar cómo una estructura dinámica como la lista enlazada puede convertirse en una estructura factible para trabajar en la "inteligencia" de la computadora, gracias a la posibilidad que tiene de guardar todas las combinaciones posibles.

Otro ejemplo sencillo puede ser el clásico solitario, donde iremos acomodando las cartas en distintas "listas". Si deseamos dirigirnos hacia un ámbito más profesional, necesitaremos cargar diferentes tareas en un listado en donde se pueda ir resolviendo cada una de ellas, de modo tal que no necesite un orden específico. Por ejemplo:

Hacer inventario	Chequear cantidad de productos	Instalar software de oficina	Realizar mantenimiento

Debido a que en esta lista de tareas es posible ir agregando o reduciendo el número de tareas, necesitaremos una estructura dinámica que se adapte al espacio en memoria.

Pila

Dentro de esta estructura, la acción más apropiada es apilar objetos/elementos. Por ejemplo: apilar un naipe sobre otro, apilar un plato sobre otro, apilar recibos, etc.

Cuando realizamos estas acciones de apilar, creamos una **pila** de objetos/elementos. Si volcamos esto a las estructuras de datos, veremos el tipo de dato abstracto llamado **Pila** y sus utilidades en la programación. Un ejemplo de estas pilas de acciones sobre un programa puede ser la acción de deshacer de procesadores de textos y planillas de cálculo, o el historial de acciones de software editor de imágenes.

Para utilizar este tipo de estructura, debemos tener en cuenta definirla de la siguiente forma:

PSEUDOCÓDIGO	C++

Estructura LIFO
  Variable Dato tipo numérica
  Variable Siguiente tipo LIFO
Fin estructura
	struct LIFO
	{
	  int Dato;
	  struct LIFO *sig;
	};

En la programación, este tipo de estructuras es conocido como **LIFO** (Last In First Out), que significa "último en entrar y primero en salir". Como podemos observar en el código del ejemplo anterior, la definición es idéntica a la creación de un nodo.

Figura 41. Representación de una pila tomándola como una estructura declarada en memoria.

Esta representación puede verse claramente en la **Figura 41**.

Ahora que conocemos cómo declarar una pila, es importante que repasemos las diferentes operaciones que podemos llevar a cabo sobre ellas, para luego aplicarlas a un ejemplo práctico:

OPERACIONES	
▼ ACCIÓN	▼ UTILIDAD
Inicializa Pila	Útil para dar un valor inicial a la estructura.
PilaVacia	Notifica si hay o no elementos en la pila.
Push	Sirve para "empujar" elementos en la pila.
Pop	Permite retirar elementos de la pila.
Listar y Buscar	Muestra todos los elementos y busca un dato específico.

Tabla 3. Operaciones en una pila.

Aunque el proceso de armado y manejo de una pila parezca sencillo, debemos prestarle mucha atención a su forma algorítmica de desarrollarlo. Para esto, a continuación veremos las operaciones que podemos realizar en dicha estructura.

Crear una pila

Si la estructura no está declarada o está vacía, no podremos hacer uso de ella hasta desarrollar lo siguiente:

PSEUDOCÓDIGO	C++
Estructura LIFO	struct LIFO
Variable Dato tipo numérica	{
Variable Anterior tipo LIFO	int Dato;
Fin estructura	struct LIFO *sig;
	};
Variable tope tipo LIFO = nulo	Struct LIFO *tope;
Variable base tipo LIFO = nulo	Struct LIFO *base;
Variable Nuevo tipo LIFO	Struct LIFO *nuevo;
Variable AuxDato tipo texto	varAux int;
Inicio CrearPila	void CrearPila()

```
Si tope = nulo entonces
        Nuevo = new LIFO
Si Nuevo <> nulo entonces
Leer AuxDato
Nuevo.Dato = AuxDato
Nuevo. Anterior = nulo

tope = nulo
base = nulo
Fin Si
Si no
ESCRIBIR "La pila está creada, debe insertar
un dato"
Fin Si
Fin CrearPila
```

```
{
        if(tope==NULL)
{
nuevo = new LIFO;
if(nuevo!=NULL)
{

        Cin>>varAux;

        Nuevo->dato=varAux;

        Nuevo->siguiente=tope;


        tope=nuevo;

        base=nuevo;
}
}

else
{
        cout<< "La pila está creada, debe
insertar un dato";
}
}
```

Teniendo en cuenta que la palabra **new** creará un nodo en memoria, veamos un ejemplo de cómo iniciaría nuestra pila, en la **Figura 42**.

MEMORIA RAM

La memoria RAM (*Random Access Memory*) es considerada la memoria principal de un equipo informático. Esto se debe a que es la encargada de brindar el soporte para la carga de software y ayudar a todo el procesamiento de información, por parte del microprocesador.

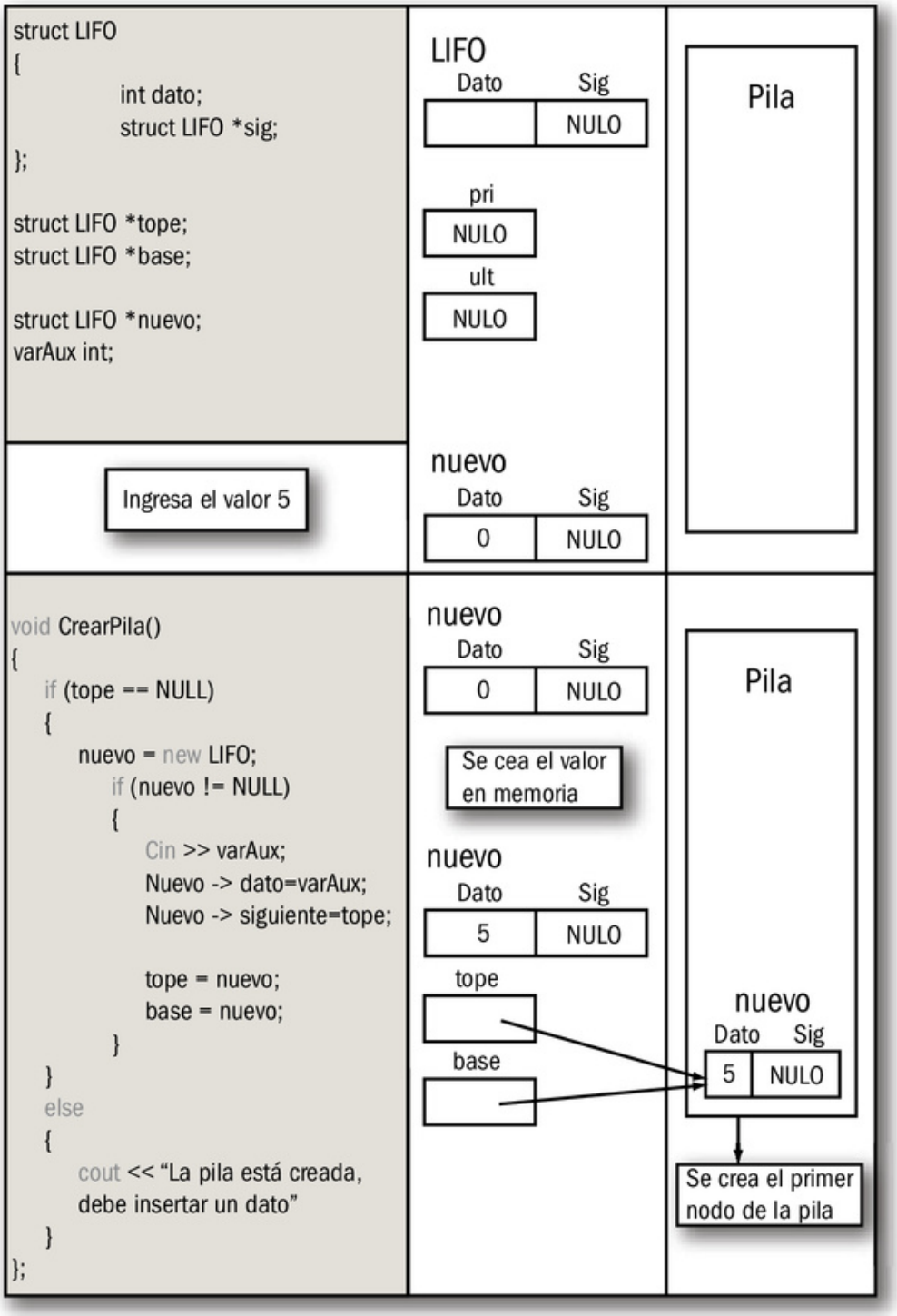


Figura 42. En este esquema podemos ver la representación de cómo se crea una pila en código C++ y cómo se vería gráficamente.

Insertar en una pila (push)

Para insertar datos en una pila, es importante tener en cuenta a la variable apuntadora **tope**, es decir que la pila puede moverse solo por uno de sus extremos.

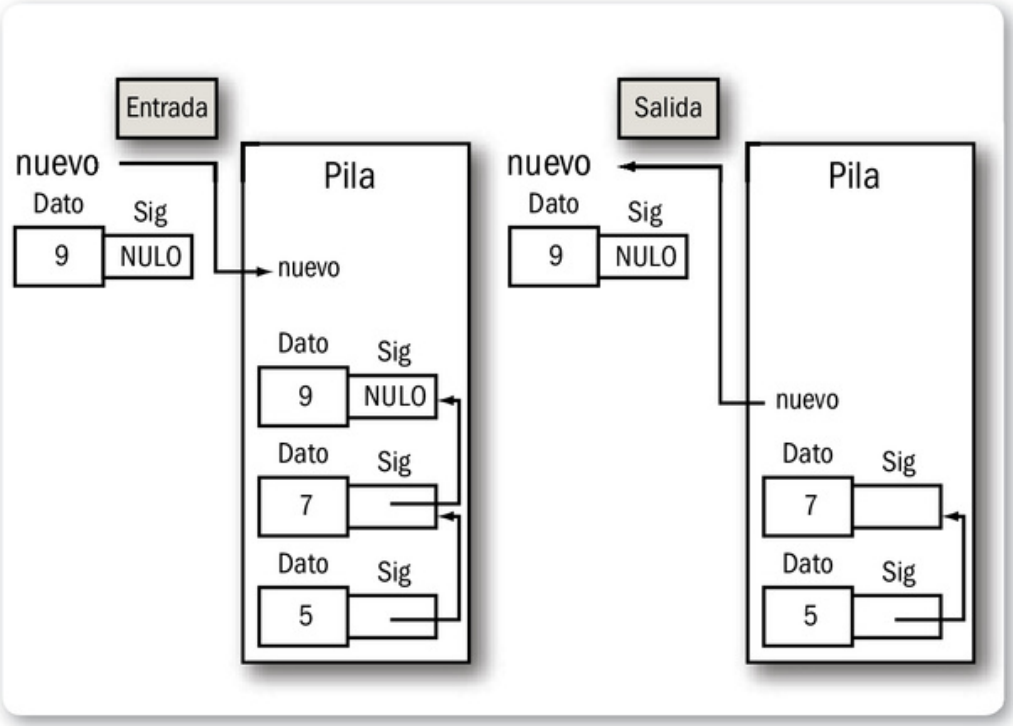


Figura 43. Representación gráfica de lo que sucede al insertar y eliminar en una pila.

PSEUDOCÓDIGO	C++
...	...
...	...
Inicio InsertarEnPila	void Push()
Si base <> nulo entonces	{
nuevo = new LIFO	if(base!=NULL)
Si nuevo <> nulo entonces	{
Leer AuxDato	nuevo = new LIFO;
nuevo.Dato = AuxDato	if(nuevo!=NULL)
nuevo. Anterior = tope	{

```
tope = nuevo
Fin Si
Si no
ESCRIBIR "La pila no fue creada, debe
crearla"
Llamar CrearPila
Fin Si
Fin InsertarEnPila
```

```
Cin>>varAux;
            Nuevo->dato=varAux;

            Nuevo->siguiente=tope;

            tope=nuevo;

}
}
Else
{
            cout<< "La pila no fue creada,
debe crearla";
   CrearPila();
}
}
```

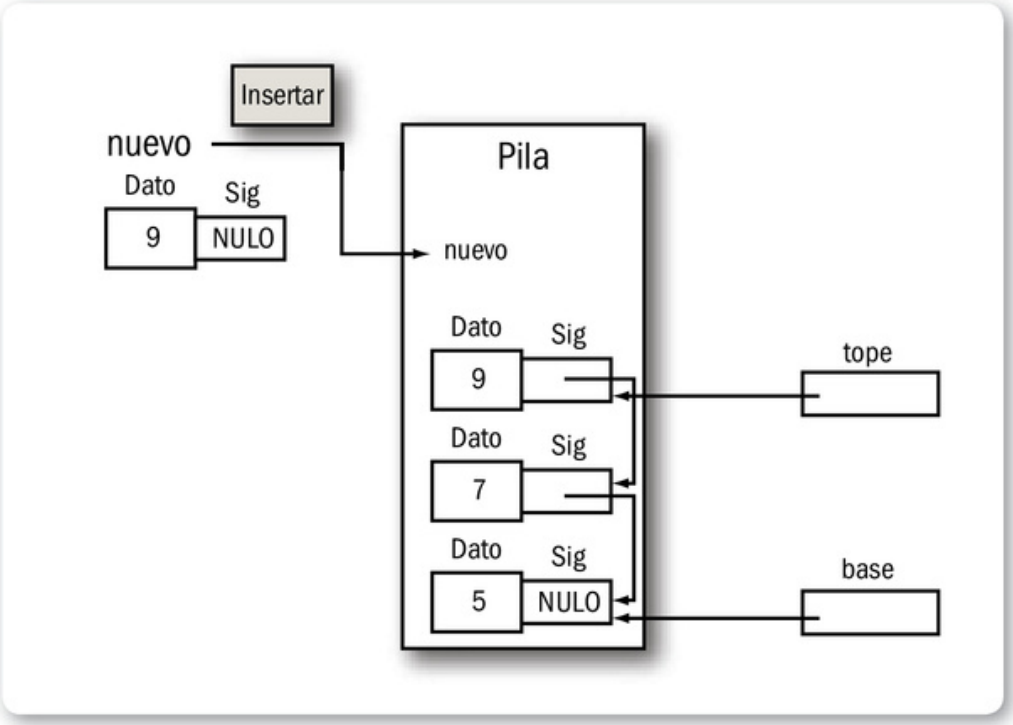


Figura 44. En esta imagen podemos ver la representación gráfica de lo que sucede en la inserción de una pila.

Eliminar en una pila (pop)

Para suprimir un nodo debemos utilizar un apuntador auxiliar (por ejemplo **P**) que guarde la dirección del **tope** de la pila, y otro auxiliar que almacene el dato que contiene el nodo a eliminar. El apuntador **tope** deberá indicar la dirección del nodo anterior al que va a ser eliminado. Luego, para eliminar el nodo, utilizamos la función **delete** e indicamos qué valor se quita de la pila. Veamos el código:

PSEUDOCÓDIGO

```
...
...
Variable P tipo LIFO

Inicio EliminarEnPila
Si base= nulo entonces
ESCRIBIR "La pila no fue creada, debe
crearla"
Si no
Si (tope=nulo) y (base<>nulo) entonces
Eliminar (base)
Else
P=tope
VarAux=tope.Dato
tope=P.Anterior

Eliminar(P)

ESCRIBIR "El dato que acaba de salir de la
pila es: " & VarAux

Fin Si
Fin Si
Fin EliminarEnPila
```

C++

```
...
...
struct LIFO *P;
void Pop()
{
        if(base==NULL)
{
cout<< "La pila no tiene elementos";
}
Else
{
if(tope==NULL) and (base!=NULL)
{
delete(base);
}
else
{
P=tope;
VarAux=tope.Dato;
tope=P.Sig;

delete(P);

cout<< "El dato que acaba de salir de la pila
es: " + VarAux
}
}
}
```

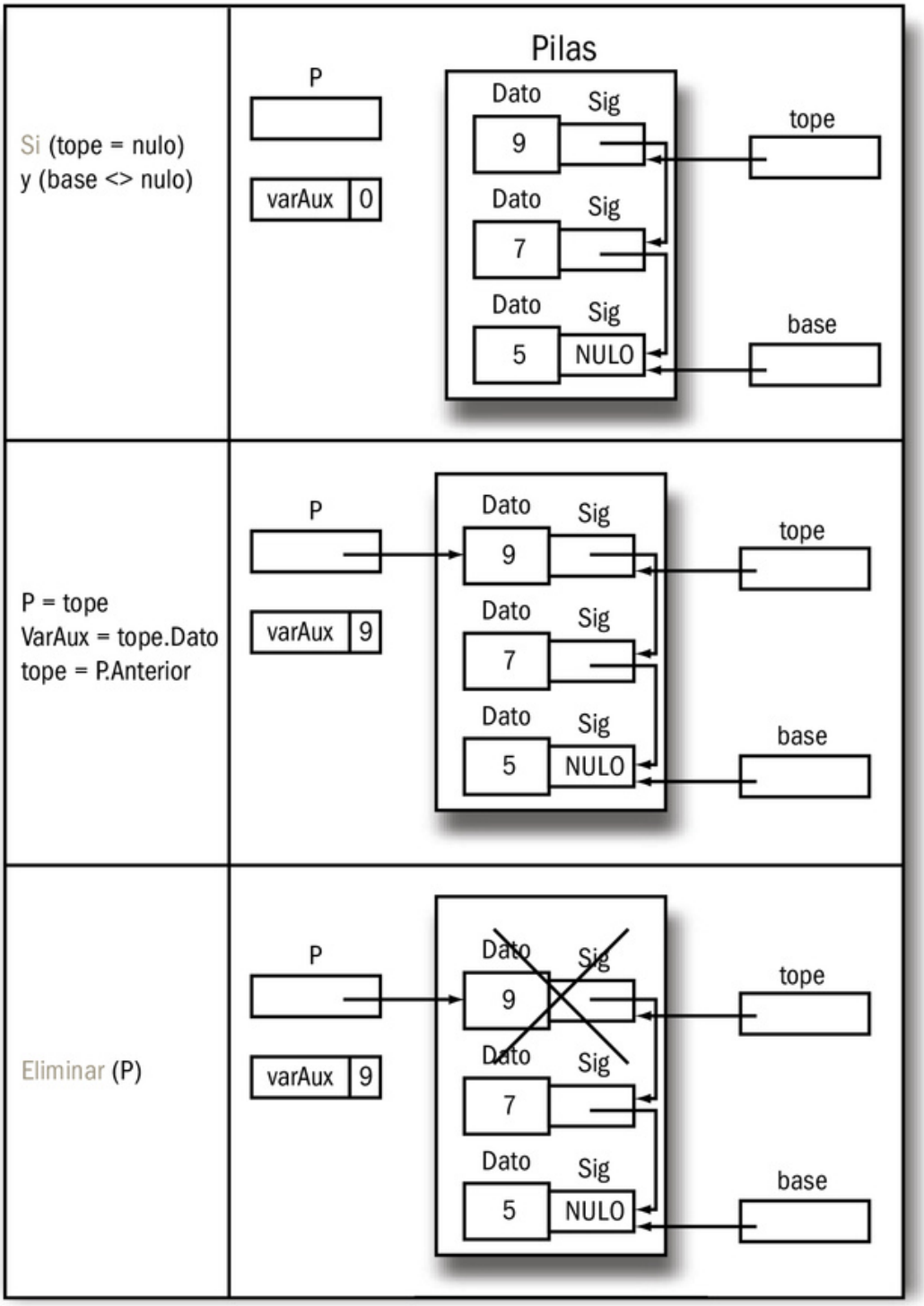


Figura 45. En esta imagen podemos ver la representación gráfica de lo que sucede al eliminar un nodo de una pila.

Listar los elementos de una pila

El listado de los elementos se realiza en forma secuencial, desde el último hasta el primero, utilizando siempre una variable auxiliar que es la que va leyendo cada uno de los nodos.

PSEUDOCÓDIGO	C++
...	...
...	...
	void Listar()
Inicio ListarPila	{
Si base<>nulo entonces	if(base!=NULL)
P=tope	{
Mientras P<>nulo	P=tope;
escribir P.Dato	While(P!=NULL)
P=P.Anterior	{
Fin mientras	cout<< P.Dato;
Sino	P=P.sig;
Escribir "No hay elementos para listar"	};
Fin Si	}
Fin ListarPila	Else
	{
	cout<< "No hay elementos para listar"
	}
	}
	}

Buscar elementos en una pila

Para buscar un elemento dentro de una pila, necesitamos utilizar un algoritmo muy similar al que aplicamos antes para listar los elementos. En este código, es importante que agreguemos un segmento condicional dentro de la repetitiva.

Hasta aquí hemos repasado las diferentes operaciones que podemos realizar sobre una estructura de dato **PILA**. A continuación vamos a trabajar en el empleo de una de ellas, aplicado a lo que se conoce como **máquina postfija**. Para darnos una idea más clara, podemos relacionarlo con las operaciones aritméticas de una calculadora.

Vector en donde tendremos datos cargados:

PSEUDOCÓDIGO	C++

...
...
 	
Inicio BuscarPila
LEER VarAux
Si base<>nulo entonces
P=tope
Mientras (P<>nulo) y (VarAux <> P.dato)
P=P.Anterior
Fin mientras
 	
Si (P<>nulo)
Escribir "Dato no encontrado"
Sino
Escribir "Encontramos el dato!!!"
Fin si
Sino
Escribir "No hay elementos para listar"
Fin Si
Fin BuscarPila
	...
	...
	void Buscar()
	{
	cin>>VarAux
	if(base!=NULL)
	{
	P=tope;
	While(P!=NULL)and(VarAux!=P.dato)
	{
	P=P.sig;
	};
	if(P<>NULL)
	{
	cout<< "Dato no encontrado"
	}
	else
	{
	cout<< "Encontramos el dato!!!"
	}
	}
	Else
	{
	cout<< "No hay elementos para listar"
	}
	}
	}

3	4	2	4	^	*	+
0	1	2	3	4	5	6

La tarea del algoritmo es recorrer el **vector**, tomar los valores que son numéricos y cargarlos en una pila determinada. De esta forma, se constituye el paso 1 que podemos observar en la **Figura 46**.

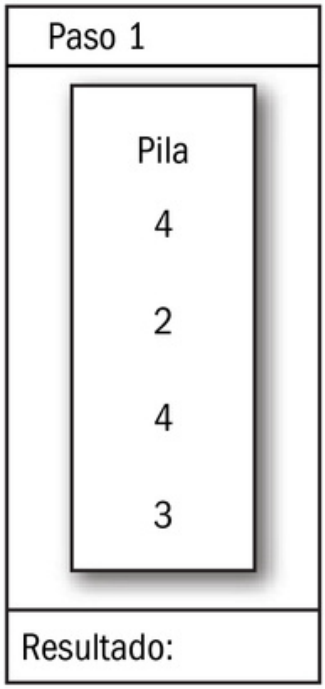


Figura 46. Representación de la pila y sus datos, utilizada para la máquina postfija – paso 1.

Una vez que hayamos cargado la pila con los valores numéricos, debemos recorrer el vector y comenzar a ejecutar los operadores aritméticos, como + - * / ^. Luego debemos tomar de la pila dos valores numéricos para ejecutar dicho operador y así llegar al paso 2.

Como podemos ver en la **Figura 47**, los dos valores que están al tope de la pila son tomados para llevar a cabo la operación: 4 ^ 2. El resultado se almacena en la pila, y se siguen tomando dos valores numéricos continuando con la siguiente operación en el paso 3. Así sucesivamente, el resultado del paso 3 es almacenado en la pila y se ejecutan los resultados, para dar el resultado final en el paso 4.

ÁRBOL BINARIO

El árbol binario de búsqueda con su estructura en árbol permite que cada nodo apunte a otros dos: uno que lo precede en la lista y otro que lo sigue. Los nodos apuntados pueden ser cualesquiera de la lista, siempre que satisfagan esta regla básica: el de la izquierda contiene un valor más pequeño que el nodo que lo apunta, y el nodo de la derecha contiene un valor más grande.

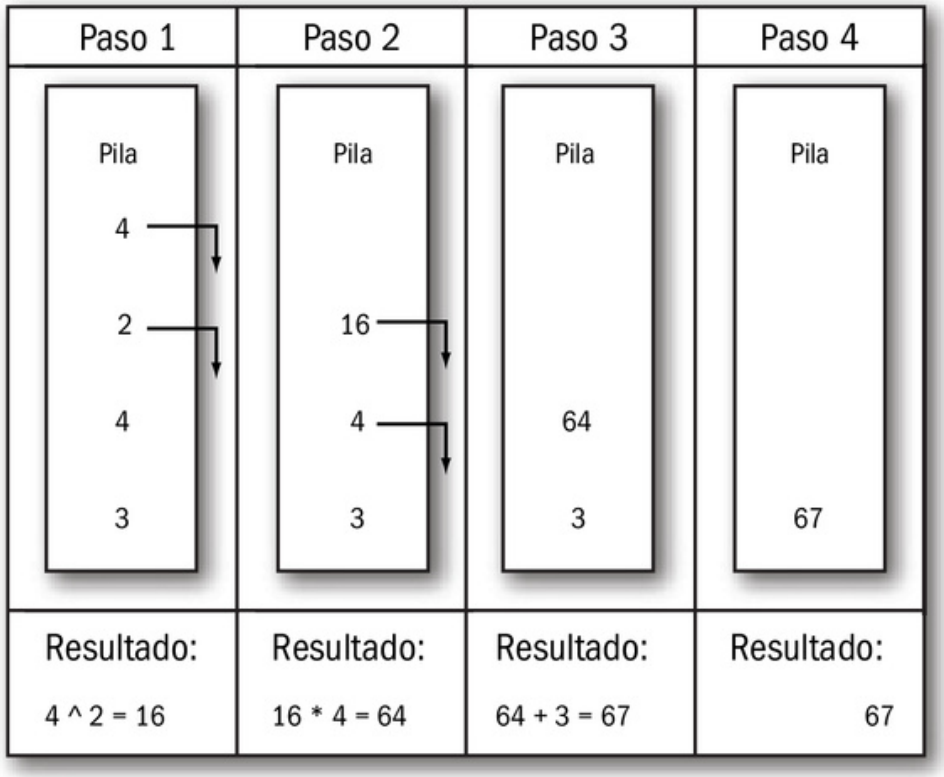


Figura 47. Representación de lo que está sucediendo internamente en la máquina postfija – pasos 2, 3 y 4.

Cola

Esta estructura de datos representa la agrupación de elementos que quedan en espera hasta ser utilizados en un orden determinado. Si lo llevamos a nuestra vida cotidiana, podemos ver ejemplos muy claros, como cuando hacemos la fila para pagar en el supermercado o sacar una entrada para ir al cine.

Bajando estos ejemplos a la informática, veremos que nuestras aplicaciones utilizan a diario esta estructura. Por ejemplo, la **cola** de impresión en el spooler (memoria) que se genera en el envío de múltiples archivos por imprimir. El primer archivo es el que sale impreso y, luego, salen los siguientes.

Para conocer en detalle cómo funciona internamente este concepto, debemos tener en cuenta que este tipo de estructura de lista también es conocida como lista **FIFO** (*First In First Out*): el primero en entrar es el primero en salir.

Para utilizar este tipo de estructura, definimos un nodo de la misma forma en que venimos trabajando:

PSEUDOCÓDIGO	C++
Estructura FIFO	struct FIFO
Variable Dato tipo numérica	{
Variable Siguiente tipo FIFO	int Dato;
Fin estructura	struct FIFO *sig;
	};
Estructura primero	struct FIFO *primero;
Estructura ultimo	struct FIFO *ultimo;

A continuación, veamos un ejemplo gráfico que nos indique las variables **primero** y **último**, teniendo en cuenta que su funcionamiento se da por orden de entrada.

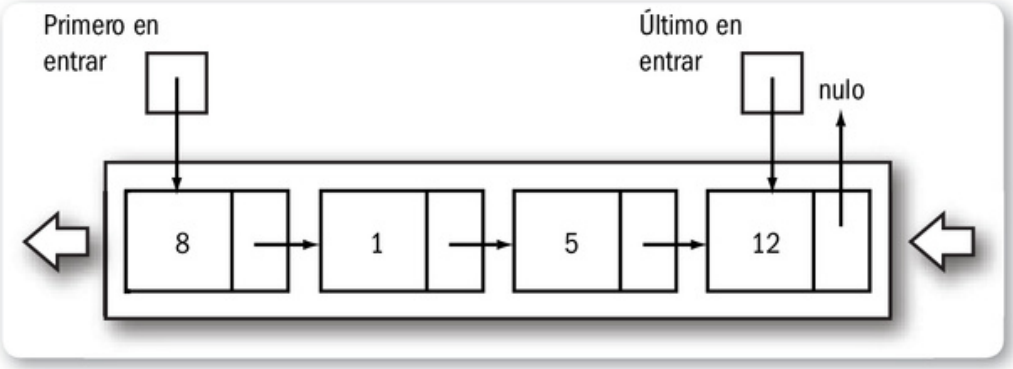


Figura 48. Representación del funcionamiento de una cola y los diferentes elementos que debemos tener en cuenta.

La cola es una de las estructuras dinámicas que se caracterizan por su estructura lineal y porque el primer dato ingresado será el primero en salir. Tenemos dos apuntadores en la figura anterior, que se llaman

primero y **ultimo**. Las flechas indican por dónde entran y salen de **FIFO**.

Debemos tener en cuenta que si la estructura cola no existe o está vacía, esta no existirá. Esto debe controlarse por código.

Crear una cola

Como vimos anteriormente, si una cola está vacía, es necesario crearla para poder ingresar los datos. Para hacerlo, debemos realizar el algoritmo que veremos a continuación.

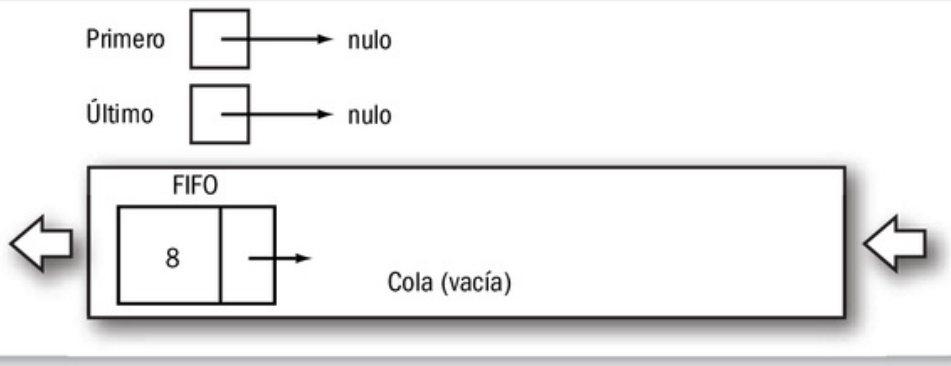


Figura 49. Representación de una cola vacía y el estado en que está cada elemento que la compone.

Cuando no se ingresan datos en la estructura, las variables auxiliares **primero** y **ultimo** (que almacenan la dirección de memoria del primero y último nodo) no tienen ninguna dirección, por lo que su valor es nulo.

PSEUDOCÓDIGO	C++
...	...
...	...
Algoritmo CrearCola	void CrearCola()
Inicio	{
Variable Nuevo tipo FIFO	struct Nodo *Nuevo;
Variable VarAux tipo numérico	int VarAux;
	if(primero!=NULL)
Si primero= nulo entonces	{
Nuevo = new FIFO	nuevo = new FIFO;

```
Si Nuevo <> nulo entonces
Leer VarAux
Nuevo.Dato = VarAux
Nuevo.Sig = Nulo
primero = Nuevo
ultimo=Nuevo
Si no
Escribir "No existe memoria" mensaje.
Fin Si
Si no
Escribir "La cola ya está creada"
Llamar Insertar()
Fin Si
Fin
```

```
if(nuevo!=NULL)
{
        Cin>>VarAux;
        Nuevo->dato=varAux;
        Nuevo->siguiente=NULL;
        primero=Nuevo;
        ultimo=Nuevo;
}
}
Else
{
        cout<< "La cola ya fue creada";
   Insertar();
}
}
```

Revisemos lo que acabamos de hacer en el código:

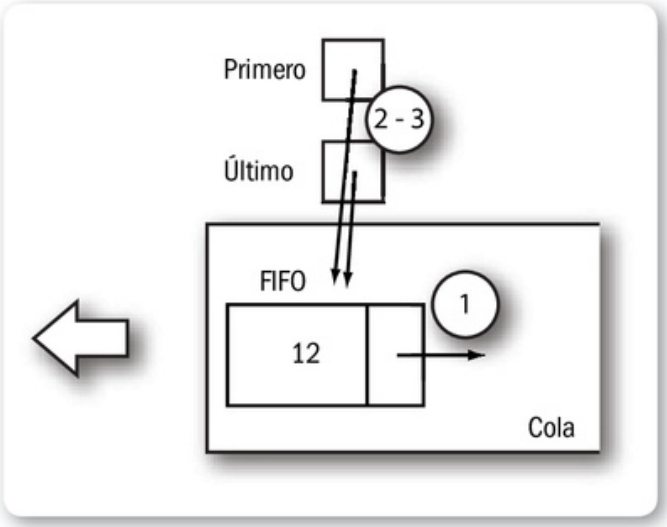


Figura 50. Representación de lo que sucede en la cola vacía al ejecutar el código anterior.

1. Tomamos **FIFO.siguiente** y hacemos que apunte a nulo.
2. Indicamos que el puntero **primero** apunte a **FIFO**.
3. Indicamos que el puntero **último** también apunte a **FIFO**.

Insertar en una cola

Así como venimos trabajando en la inserción de nodos en estructuras, en este caso el algoritmo es similar y solo debemos tener en cuenta que se insertará al final de ella.

Iniciaremos con: un nodo por insertar, un puntero que apunte a él y una cola no vacía (donde los punteros **primero** y **ultimo** no serán nulos).

Ahora veamos el algoritmo y una figura representativa en la **Figura 51**.

PSEUDOCÓDIGO

```
...
...
Algoritmo InsertarCola
Inicio
Variable Nuevo tipo FIFO
Variable VarAux tipo numérico
Nuevo = new FIFO

Si Nuevo <> nulo entonces
Leer VarAux

Nuevo.Dato = VarAux
Nuevo.Sig = Nulo
Ultimo.siguiente = Nuevo
ultimo=Nuevo
Si no
Escribir "No existe memoria" mensaje.
Fin Si
Fin
```

C++

```
...
...
void InsertarCola()
{
        struct Nodo *Nuevo;
        int VarAux;
nuevo = new FIFO;
if(nuevo!=NULL)
{

        Cin>>VarAux;

        Nuevo->dato=varAux;

        Nuevo->siguiente=NULL;

        Ultimo.sig=Nuevo;

        ultimo=Nuevo;
}
else
{
```

```
cout<< "No hay memoria suficiente";
}
}
```

Revisemos lo que acabamos de hacer en el código:

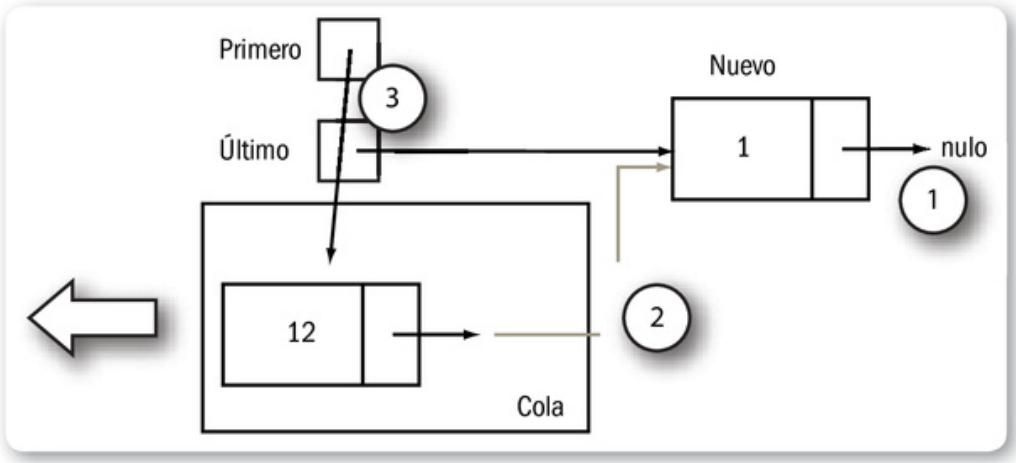


Figura 51. Representación de lo que sucede al insertar un elemento en la estructura.

1. Hacemos que **Nuevo.siguiente** apunte a nulo.
2. Luego que **ultimo.siguiente** apunte a **Nuevo**.
3. Y actualizamos **ultimo**, haciendo que apunte a **Nuevo**.

Eliminar elementos de una cola

Hay dos escenarios que podemos considerar para eliminar un elemento de la cola: que haya un solo elemento o que haya varios.

Es importante tener en cuenta que, para eliminar un elemento, este debe ser el primero que entró en la cola.

PSEUDOCÓDIGO	C++
...	...
...	...
Algoritmo Eliminar	void Eliminar()

```
Variable AuxNodo tipo FIFO
Inicio
Si Primero =nulo
Escribir "No hay elementos para eliminar"
Si no
Si (Primero = Ultimo)
Eliminar (Primero)
Primero = nulo
Ultimo = nulo
Si no
AuxNodo = primero.Siguiente
Eliminar (Primero)
Primero = AuxNodo
Fin Si
Fin si
Fin
```

```
{
        struct Nodo *AuxNodo;
if(primero!=NULL)
{
        cout<< "No hay elementos para
eliminar";
}
else
{

        if(primero==ultimo)

        {

        Delete(primero);

        primero=NULL;

        ultimo= NULL;

        }

        else

        {

        AuxNodo=primero->sig;

        Delete(primero);


        primero=AuxNodo;

        }
}
}
```

En el gráfico de la **Figura 52** podemos ver la representación de cómo sería el proceso del código anterior.

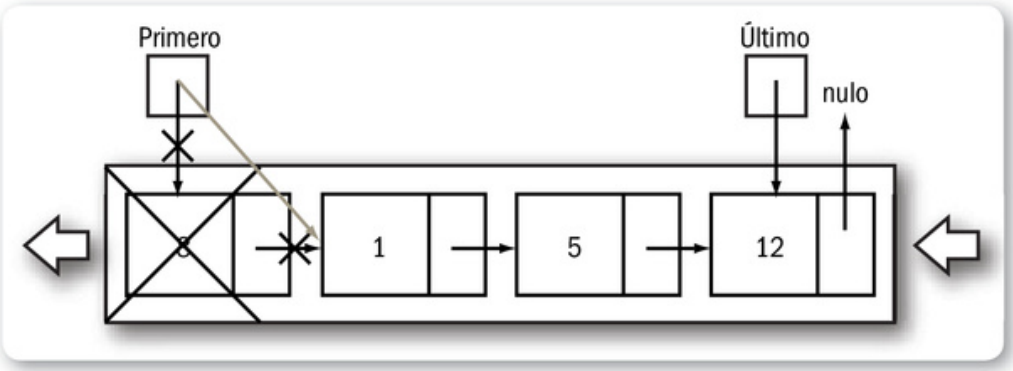


Figura 52. Representación de lo que sucede cuando se elimina un elemento de la estructura.

Revisemos algunas partes del código anterior:

El primer condicional **SI (Primero = Ultimo)** controla si la cola tiene un solo elemento y, de ser así, lo elimina. Continúa con el **SI anterior**; en caso de entrar, asignamos a **AuxNodo** la dirección del segundo nodo de la pila: **primero.siguiente**. Liberamos la memoria asignada al primer nodo, el que queremos eliminar, indicando el apuntador **primero** y, por último, asignamos a **primero** la dirección que quedó almacenada en **AuxNodo**.

Listar los elementos de una cola

En caso de tener que listar los elementos de una cola, debemos saber que los elementos no son secuenciales, sino que debemos trabajar como en los algoritmos de recorrido anteriores. Vamos a necesitar un valor auxiliar que almacene la posición de los nodos que vamos visitando.

COLA VS. PILA

La diferencia en estas estructuras radica, principalmente, en cuál elemento sale primero. En las colas, el primero en entrar es el primero en salir. En las pilas, los primeros elementos que entran son los últimos que salen; también los elementos que se insertan y eliminan pertenecen al mismo extremo.

PSEUDOCÓDIGO	C++
...	...
...	...
Algoritmo Listar	void Listar()
Variable VarAux tipo FIFO	{
Inicio	struct Nodo *AuxNodo;
Si (Primero <> Nulo) entonces	if(primero!=NULL)
VarAux = Primero	{
Mientras VarAux <> nulo	
Escribir VarAux.Dato	Cout<<"Lista de COLA: ";
VarAux = VarAux.Siguiente	
Fin Mientras	while(AuxNodo<>NULL)
Si no	
Escribir "No hay elementos"	{
Fin Si	
Fin	Cout<<AuxNodo.Dato;
	AuxNodo=AuxNodo.Sig;
	}
	}
	else
	{
	Cout<<"No hay elementos";
	}
	}

ESTRUCTURA GRAFO

Esta estructura consiste en un conjunto de nodos, también llamados vértices, y un conjunto de arcos o aristas que establecen relaciones entre los nodos. El concepto de este tipo de datos abstracto desciende directamente del concepto matemático de grafo. Por ejemplo: una lista lineal puede ser vista como un grafo, donde cada nodo está relacionado con otro nodo distinto de él.

Como podemos ver en los ejemplos de código, la variable auxiliar jugará un papel fundamental, ya que irá moviendo su puntero por cada nodo y mostrando la información que haya en ellos.

Buscar elementos en una cola

Aprovechando el código anterior de ejemplo, solo debemos ajustar con una condición el recorrido de los elementos en una estructura cola.

Veamos cómo sería el algoritmo:

PSEUDOCÓDIGO

```
...
...
Algoritmo Buscar
Variable VarAux tipo FIFO
Variable Aux tipo numerico

Inicio
Si (Primero <> Nulo) entonces
LEER Aux
VarAux = Primero
Mientras (VarAux <> nulo) y (VarAux.
dato<>Aux)
VarAux = VarAux.Siguiente
Fin Mientras
Si (VarAux = nulo) entonces
 Escribir "No encontramos el elemento"
Si no
 Escribir "Dato encontrado!!!"
Fin Si
Si no
Escribir "No hay elementos"
Fin Si
Fin
```

C++

```
...
...
void Buscar()
{
        struct Nodo *AuxNodo;
        int Aux;
if(primero!=NULL)
{

        cin>>Aux;

        AuxNodo=primero;

        while(AuxNodo<>NULL)and
(AuxNodo.dato<>Aux)

        {

        AuxNodo=AuxNodo.Sig;

        }

        if(AuxNodo=NULL)
{

        cout<<"No se encontro el
```

```
elemento buscado";
}

        else
{

         cout<<"\Encontramos el elemento
buscado!!";
}
        }
else
        {

        Cout<<"No hay elementos";
        }
}
```

Como podemos ver, el uso de colas de espera puede reconocerse en distintos tipos de negocios. Por ejemplo: se utiliza cola cuando los bancos imprimen tickets con los números de atención, para que los clientes sean llamados por medio de un monitor.

También podemos utilizar una cola para la espera de diferentes tareas que deben ser ejecutadas, por ejemplo, tareas de mantenimiento en un sistema operativo: 1-limpieza de RAM, 2-Limpieza archivos temporales, etc.

RESUMEN

Más allá de las estructuras estáticas de arreglos (vector/matriz), ahora hemos conocido otras estructuras dinámicas de almacenamiento en memoria y analizado cuáles son sus diferentes usos. También vimos las primeras estructuras dinámicas de lista manipuladas a través de distintas acciones, tales como: crear, insertar, eliminar y recorrer. En todos los casos, conocimos la forma de la sintaxis y los algoritmos por utilizar, concluyendo con algunas posibles implementaciones sobre dichas estructuras. Desde las listas se puede comprender mejor el uso de: listas enlazadas, doblemente enlazadas, pilas y colas, estructuras que ahora podemos diferenciar en el uso del software cotidiano.

Actividades

TEST DE AUTOEVALUACIÓN

1. ¿Qué son las estructuras dinámicas y cuáles son las características de una lista?
2. ¿Qué es un nodo y cómo sabemos si tenemos memoria para crearlo?
3. Compare una lista enlazada simple con una doblemente enlazada.
4. ¿Por qué el recorrido de la lista no se hace en forma secuencial?
5. ¿Cómo identificar el inicio o final de una lista?
6. ¿Qué algoritmo se utiliza para borrar un nodo ubicado en medio otros?
7. ¿Qué operaciones se pueden realizar en una pila?
8. ¿Es posible listar el contenido completo de una pila?
9. ¿A qué se llama LIFO y FIFO?
10. ¿Es posible eliminar el último elemento de una cola?

ACTIVIDADES PRÁCTICAS

1. Desarrolle una función que recibe una lista de enteros L y un número entero n, de forma que borre todos los elementos que tengan este valor.
2. Cree los valores de dos listas de enteros ordenados de menor a mayor y desarrolle una función Mezcla2 para obtener una nueva lista, también ordenada.
3. Desarrolle en código (pseudocódigo o C++) una función llamada encontrarGrande que encuentre el mayor número en una lista simple. Usted debe crear los valores que contiene la lista.
4. Desarrolle un programa en C++ que cree una estructura para generar una lista doblemente enlazada y crear las funciones para: agregar, eliminar e imprimir los elementos de la lista.
5. Desarrolle un programa que pueda crear un arreglo de 10 elementos de listas enlazadas, para guardar números enteros entre 1 y 100.
   En la posición 0 del arreglo irán todos los números ingresados menores a 10; en la posición 1, todos los números ingresados mayores o iguales a 10 y menores que 20; en la posición 2, todos los números mayores o iguales a 20 pero menores que 30, etc.

*07

Normas generales en las interfaces gráficas

Luego de haber conocido en el capítulo anterior otras estructuras dinámicas de almacenamiento en memoria y analizado cuáles son sus diferentes usos, en esta sección, trabajaremos con las normas necesarias para adentrarnos en el diseño y la confección de interfaces gráficas desde el lenguaje de programación Visual Basic.

- **Normas de diseño de interfaz 336**
  - Interfaces de usuario: evolución y estado del arte actual 337
  - Fundamentos del diseño de interfaz 345
- **Interfaces de escritorio/web/móvil 349**
  - Componentes usuales - visuales 355
- **Confección de interfaces en Visual Basic 357**
- **Resumen 365**
- **Actividades 366**

Normas de diseño de interfaz

Cuando confeccionamos interfaces gráficas, debemos considerar un conjunto de normas que pueden ser estructuradas o no, dependiendo del tipo de sistemas que queramos generar.

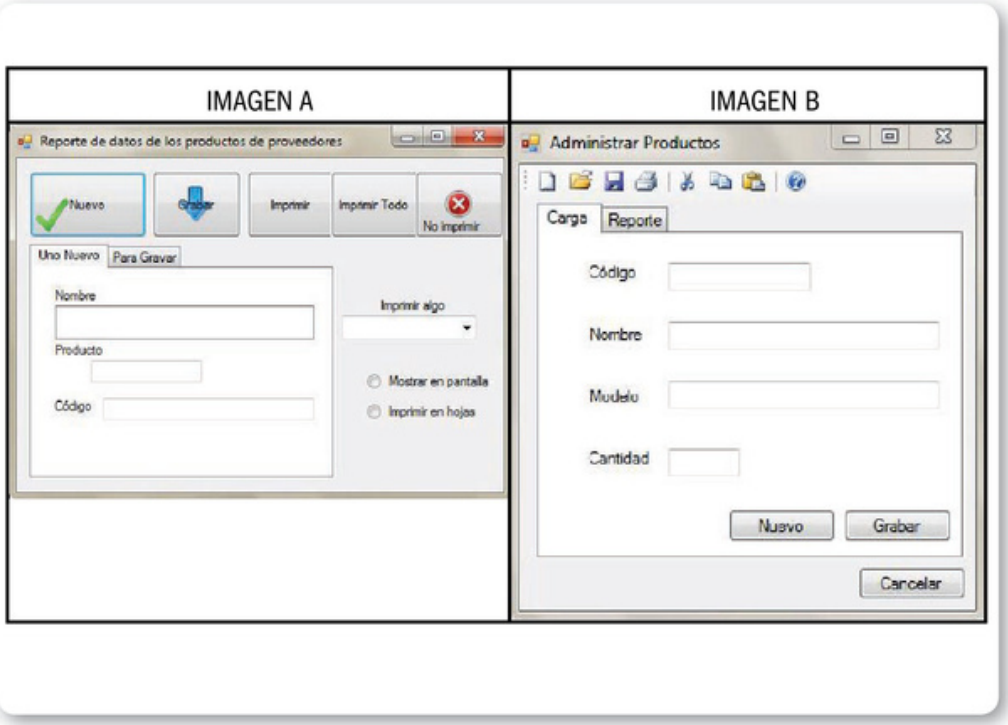


Figura 1. Interfaces gráficas que podemos encontrar en distintos entornos o dispositivos.

AGREGAR CONTROLES

Cada tipo de control tiene su propio conjunto de propiedades, métodos y eventos que lo hacen adecuado para un propósito en particular. Cuando debemos agregar repetidas veces controles del mismo tipo, podemos realizar una acción que nos facilitará este proceso: primero presionamos la tecla **CTRL** y la mantenemos así, luego seleccionamos con el botón izquierdo el control y lo agregamos en el Form las veces que deseamos; al finalizar, soltamos la tecla **CTRL**.

El conjunto de normas que veremos en los siguientes párrafos corresponde a una serie de generalidades para la buena confección de interfaces gráficas y recomendaciones que nos permitirán tener precaución al momento de diseñarlas. Podemos conceptualizar que el estudio y desarrollo del diseño de interfaz prioriza el trabajo de varias disciplinas en función de un mismo objetivo, que es comunicar y transmitir a través de un medio electrónico.

El diseño de interfaces gráficas (GUI) es un factor muy importante en el desarrollo del software, ya que es el medio que nos permitirá comunicarnos con todos los usuarios que interactúen con nuestros desarrollos. Nuestro objetivo principal es buscar la forma en que ellos puedan comunicarse con el programa de manera efectiva, obteniendo así un diseño de la interfaz confortable que no requiera esfuerzo para su comprensión. A continuación, veamos ejemplos de interfaces de aplicaciones de escritorio.

> NUESTRO OBJETIVO ES LOGRAR QUE LOS USUARIOS PUEDAN INTERACTUAR DE MANERA EFECTIVA

Como podemos observar en la imagen A de la **Figura 1**, hay cierto desorden en la interfaz gráfica a simple vista; en contraste con la imagen B, que es armónica y aparenta ser organizada. Es importante tener en cuenta que esto dependerá de cómo sea el funcionamiento de la interfaz. En el caso de esta figura, podemos suponer que se trata de administrar o cargar productos en un sistema.

A continuación, veremos diferentes interfaces que podemos encontrar en el entorno actual.

Interfaces de usuario: evolución y estado del arte actual

A lo largo de los años, los medios de comunicación han ido evolucionando de forma impresionante; por ejemplo, desde aquellos grandes equipos utilizados con tarjetas de perforación, hasta los dispositivos actuales que toman nuestro cuerpo y voz como medio de interacción con equipos electrónicos.

Repasemos ahora cuáles son los tipos de interfaces que podemos encontrar en la actualidad.

Figura 2. Interfaz de escritorio: MAC OS, sistema operativo utilizado para los dispositivos de la empresa Apple.

En la **Figura 2** podemos ver el ejemplo de una interfaz gráfica de escritorio, en este caso aplicada al sistema operativo MAC OS. En la **Figura 3**, podemos ver el modelo habitual de una interfaz de consola.

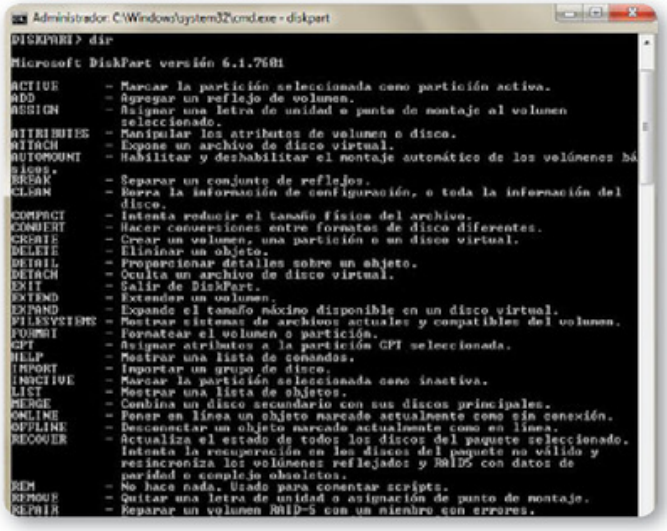


Figura 3. Interfaz de consola: MS DOS, sistema operativo empleado en modo texto.

Figura 4. Interfaz web: información en línea que se presenta a través de un portal de noticias y ofertas.

En las **Figuras 4** y **5**, veremos dos ejemplos de interfaces web que le permiten al usuario acceder a los contenidos, realizar tareas y comprender las funcionalidades.

Figura 5. Interfaz web: sitio atractivo que ofrece servicios de diseño para diferentes tipos de actividades.

Figura 6. Interfaz táctil: diferentes sistemas en los smartphones que nos permiten interactuar con los teléfonos.

En la **Figura 6** podemos ver distintos sistemas de interfaces táctiles. En el caso de la **Figura 7**, tenemos otro ejemplo de interfaz, esta vez aplicado a las consolas de videojuegos.

Figura 7. Interfaz de movimiento: Xbox 360 nos permite personalizar y utilizar diferentes dispositivos para esta interfaz.

Durante los primeros años de los equipos informáticos, los medios de interacción existían a través de tarjetas perforadas. Para darnos una idea más clara, observemos la imagen de la **Figura 8**.

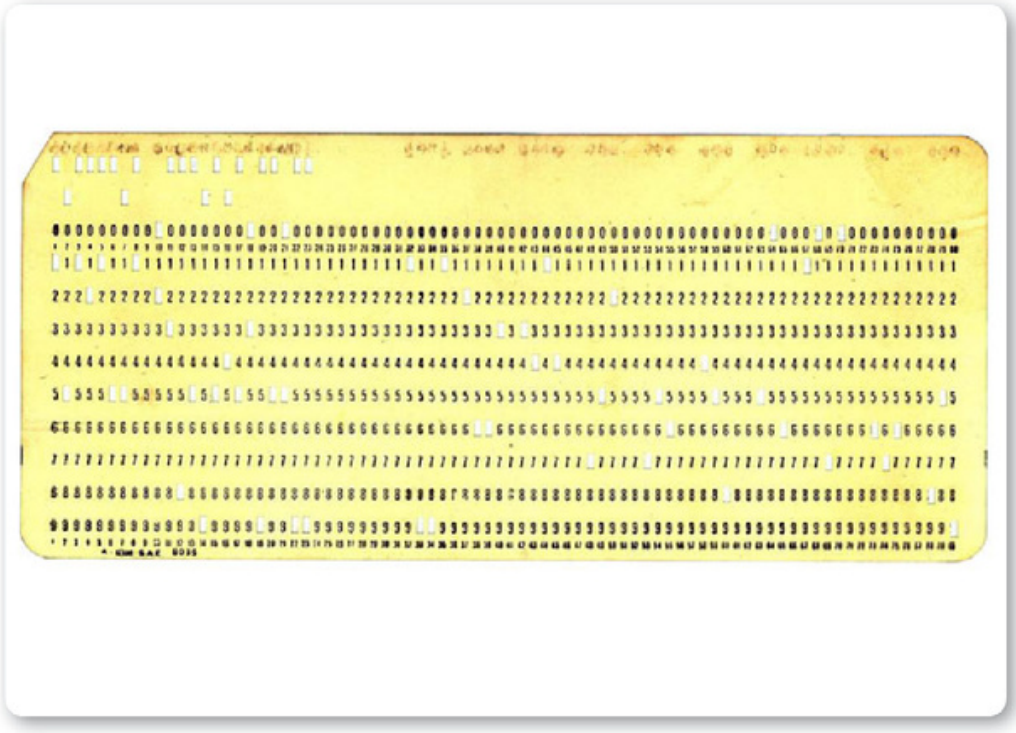


Figura 8. Tarjeta perforada binaria que se usaba hace décadas para gestionar los datos que se necesitaban procesar.

Imaginemos años atrás, cuando la forma de interactuar con los equipos informáticos era por medio de estas tarjetas, y los usuarios debían tener el conocimiento para poder "leerlas" y ver sus resultados. Es impresionante ver cómo se inició el almacenamiento de datos,

TARJETA PERFORADA

La tarjeta perforada es una lámina hecha de cartulina que contiene información a modo de perforaciones según un código binario. Fueron los primeros medios utilizados para ingresar información e instrucciones en una computadora en los años 1960 y 1970. Las tarjetas perforadas fueron usadas por Joseph Marie Jacquard en los telares que inventó.

utilizado muchas veces por bancos cuyos equipos de cómputos solían ocupar salones completos.

Continuando con equipos personales, se utilizaron interfaces de consola donde se escribía solo texto. Como podemos ver en la **Figura 3**, los sistemas operativos se cargaban en memoria, como **DOS**, y se manejaban por medio de comandos escritos con el teclado. Para ejecutar lo que deseábamos, era necesario conocer el listado de operaciones posibles.

A continuación, veamos otro ejemplo de los años 80.

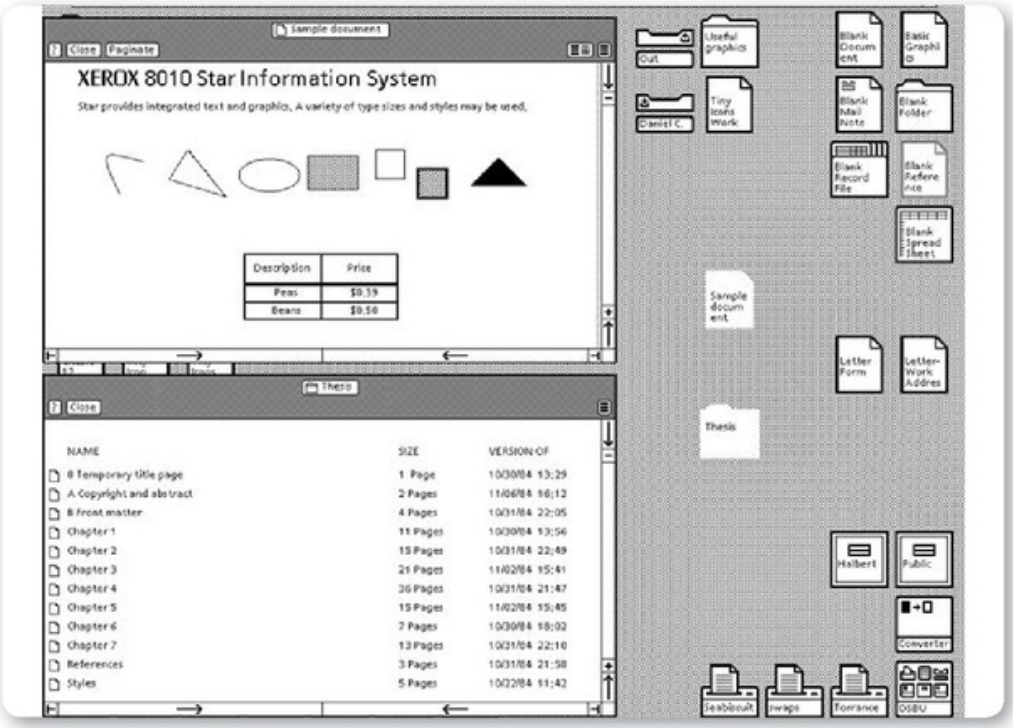


Figura 9. Xerox 8010 Star (1981) fue una de las primeras interfaces gráficas en el sistema operativo.

XEROX 8010 STAR

Fue el primer sistema que combinaba una computadora de escritorio con una serie de aplicaciones e interfaz gráfica de usuario (GUI). Se lo conoció originalmente como Xerox Star, aunque luego sufrió dos cambios en su nombre: primero a ViewPoint y, más tarde, a GlobalView.

Podemos destacar que la evolución de colores en las interfaces fue una revolución en la interacción usuario/máquina. Esto se pone de manifiesto en la **Figura 10**.

Figura 10. Amiga Workbench 1.0 (1985) fue una de las primeras interfaces en color que salió al mercado.

La evolución de interfaces sobresale en la aparición de Windows 95, un sistema operativo cuya interfaz gráfica amigable lo convirtió en uno de los más instalados en computadoras personales. Debido a ello, la mayoría de las empresas fabricantes de hardware y software tienden a desarrollar sus aplicaciones basadas en dicho sistema.

AMIGA

Sistema operativo del **Amiga**, cuya interfaz gráfica fue adelantada para su tiempo, ya que ofrecía cuatro colores (negro, blanco, azul y naranja), multitarea preventiva (adoptada recién 10 años después por otros sistemas), sonido estéreo e iconos multiestado (seleccionados y no seleccionados).

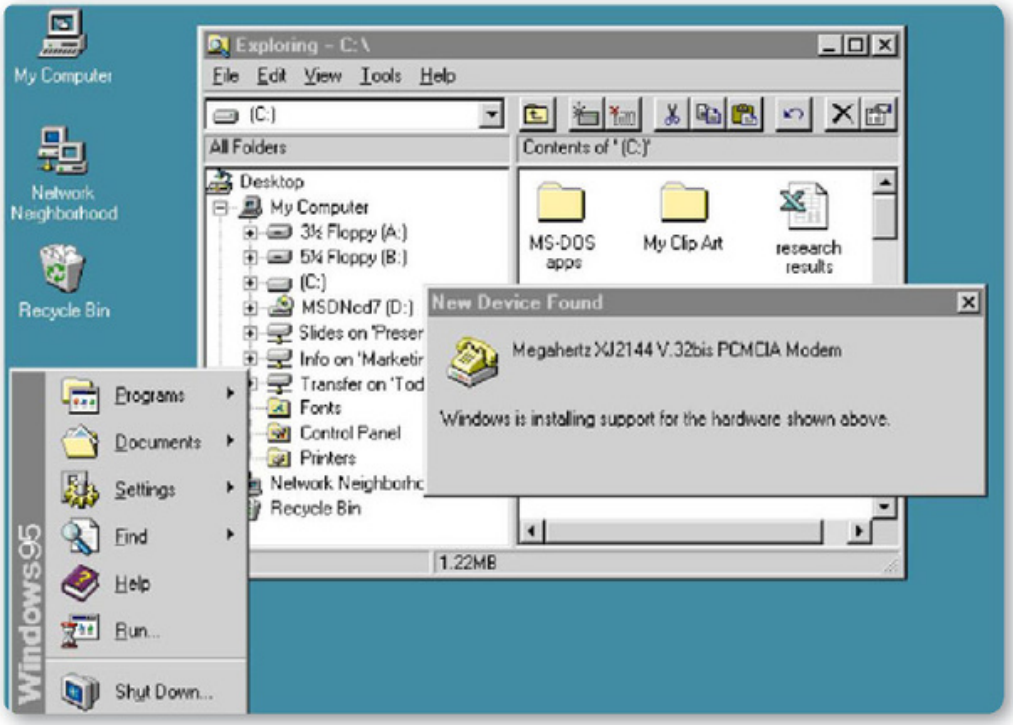


Figura 11. Windows 95 (1995) es el sistema operativo con el que Microsoft logró posicionarse en el mercado mundial.

En el transcurso del año 2000, podemos encontrar varias revoluciones de interfaces gráficas muy estéticas y fáciles de utilizar, como **Windows XP**, **Linux** y **MAC OS**.

Si lo pensamos en la actualidad, descubrimos interfaces muy simpáticas y amenas que utilizan diferentes medios para interactuar, como vimos en las **Figuras 4**, **5**, **6** y **7**. A continuación, veamos cuáles son los sistemas operativos más frecuentes del momento.

WINDOWS 95

En Windows 95 se aplicó una nueva interfaz de usuario que fue compatible con nombres de archivo largos de hasta 250 caracteres, capaz de detectar automáticamente y configurar el hardware instalado. En esta versión se incluyeron numerosas mejoras, entre ellas: se ofrecieron varios estados a los iconos, apareció por primera vez el famoso botón de **Inicio**, y surgieron otros conceptos relacionados con el aspecto visual que se conservan hasta el día de hoy.

SISTEMAS

EMPRESAS	VERSIONES	DISPOSITIVOS
Microsoft	Windows 8 http://windows.microsoft.com/es-XL/windows-8/release-preview	Computadora de escritorio/laptop Equipos portátiles (smartphone/tablet)
Apple	MAC OS X Mountain Lion www.apple.com/osx	Computadora de escritorio/laptop
Apple	iOS 6 www.apple.com/ios	Equipos portátiles (smartphone/tablet)
Linux	Varias versiones www.linux.org	Computadora de escritorio/laptop Equipos portátiles (smartphone)
Android	Android 4.1, Jelly Bean www.android.com	Equipos portátiles (smartphone)

Tabla 1. Sistemas operativos utilizados tanto en dispositivos móviles como en equipos de escritorio.

Fundamentos del diseño de interfaz

Existen varios factores que influyen en la toma de decisiones sobre el diseño de interfaz; por eso, podemos considerar las siguientes preguntas que nos ayudarán a encaminar nuestra perspectiva.

DISEÑO

PREGUNTA	DESCRIPCIÓN
¿Por qué y para qué?	Cuál es el objetivo de nuestro diseño.
¿Para quién?	Lo que vamos a diseñar puede ser para una persona o un grupo, para una tarea genérica o específica, para automatizar procesos, etc. Por eso, es fundamental saber al sector al que nos dirigimos, conocer sus habilidades y experiencias. Otro aspecto importante es el género en donde será utilizado el desarrollo: gerente general, técnico, científico, etc.

¿Con qué tipo de recursos tecnológicos cuenta el usuario?	¿Es un desarrollo para Macintosh, PC, Linux, todos? Estos datos nos ayudarán a comprender las necesidades del usuario, y así evaluar cómo presentar la información, en qué formato, qué lenguaje de programación utilizar, etc.

Tabla 2. Estos son algunos de los fundamentos importantes de diseño que debemos tener en cuenta a la hora de programar.

Estas preguntas importantes nos servirán para fundamentar nuestros diseños gráficos, darles sentido y funcionalidad. Como veíamos antes, el diseño de la interfaz debe garantizar un sentimiento de seguridad que guíe y condicione al usuario, para así brindar un control total sobre ella. Para generar estas condiciones, debemos considerar distintos recursos multimedia, imágenes, sonidos, animaciones, videos, etc., aplicándolos en su medida justa para lograr la integridad del diseño. Por ejemplo, debemos evitar caer en inadecuadas aplicaciones de los pedidos de usuario.

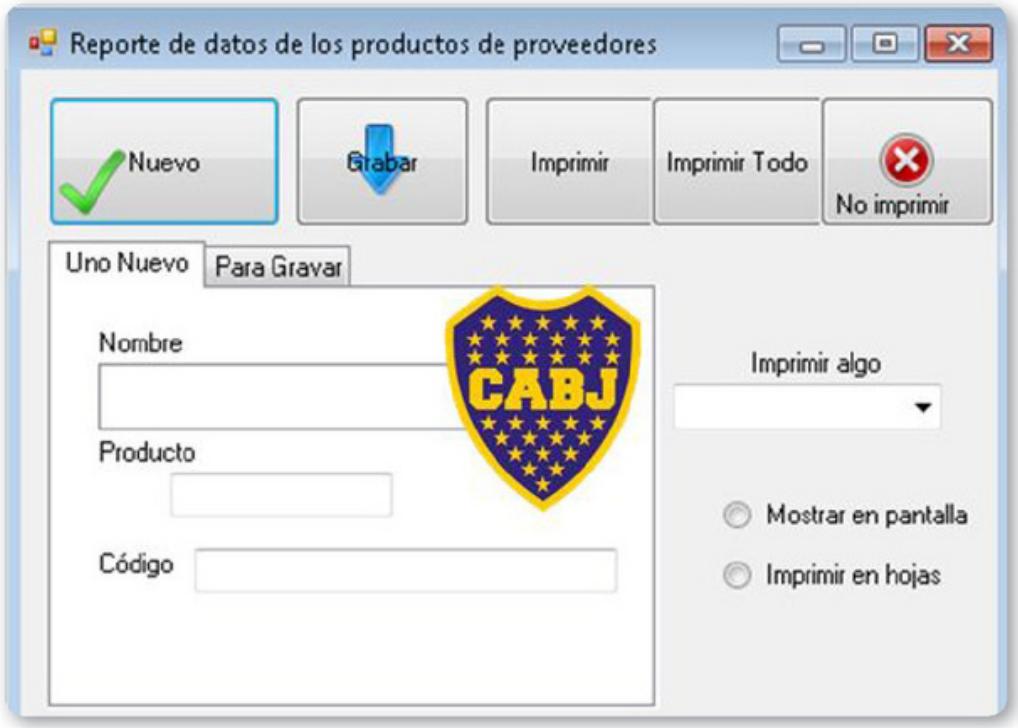


Figura 12. Una mala aplicación de concepto puede ser incluir en la interfaz gustos peculiares, como un cuadro de fútbol.

No debemos tomar todos los gustos individuales de una persona que utilizará el desarrollo, como veíamos en la **Figura 12**. Es importante tener un cierto discernimiento para saber seleccionar adecuadamente las características personalizadas.

A continuación, veamos algunos fundamentos que debemos tener en cuenta en el diseño de una interfaz gráfica.

1. **Sencillez**: evitar la saturación e inclusión innecesaria de elementos, ya que su función es acompañar y guiar al usuario.
2. **Claridad**: la información debe ser fácil de localizar. Para eso, es necesario establecer una organización, ya sea de manera lógica, jerárquica o temática. Esto es fundamental porque, cuanta más información ubiquemos en la pantalla, mayor será la distracción o confusión para el usuario.
3. **Predecibilidad**: para acciones iguales, resultados iguales.
4. **Flexibilidad**: el diseño de la interfaz debe ser claro y predecible en cuanto a la mayor cantidad posible de plataformas.
5. **Consistencia**: lograr una semejanza entre las diferentes interfaces del sistema, estableciendo una organización según la función de los elementos. Para el usuario, es importante que la interfaz sea similar en las distintas aplicaciones. Por ejemplo, en una aplicación web, la ubicación del listado de contenido suele estar del lado izquierdo.
6. **Intuición**: el usuario debe poder sentirse seguro y no tener que adivinar ni pensar cómo ejecutar las acciones. Por eso, es importante que la aplicación lo "guíe" para realizar su cometido, sin necesidad de recurrir a un mapa de ruta.

> ES IMPORTANTE QUE LOS ELEMENTOS UTILIZADOS CORRESPONDAN AL CONTENIDO

TABCONTROL

El control **TabControl** contiene páginas de fichas representadas por objetos **TabPage**, que se agregan mediante la propiedad **TabPages**. El orden de las páginas de fichas refleja el orden en que las fichas aparecen en el control. Es una herramienta muy útil para ordenar controles o información, ya que el usuario puede cambiar el objeto **TabPage** actual haciendo clic en una de las fichas del control.

a. Disminuir las interacciones: es importante reducir y simplificar la interacción del usuario, ya que su fin es el resultado que le brinda el sistema.
Por ejemplo: cuando el usuario utiliza Google, no lo hace por el gusto de hacer clic en "Buscar" ni por el atractivo de su logo. Por eso, cualquier interacción que Google incluya va a interponerse en la necesidad del usuario. Actualmente, se ha reducido la interacción gracias a la búsqueda inteligente, porque a medida que el usuario escribe sobre el buscador, van apareciendo los resultados posibles.
b. Orden y presentación: la información debe mostrarse en forma lógica, agrupando todos los campos que sean similares entre sí.
Por ejemplo: en la carga de datos personales, debemos agrupar los datos del domicilio por un lado (dirección, código postal, ciudad, etc.) y los niveles educativos alcanzados por el otro.

7. **Coherencia**: todos los elementos utilizados (textos, gráficos, colores, etc.) deben corresponder al contenido de la publicación.
8. **Buen uso de controles**: en diseño, es fundamental el uso correcto de componentes visuales, tales como: etiquetas de texto, campos de texto, listas desplegables, casillas de opción, botones de opción y grillas de resultados. Cada componente tiene un comportamiento y una utilidad que se deben alterar, para así no confundir al usuario.
a. Tamaño de los componentes: debemos cuidar el aspecto de los componentes, ya que si modificamos su tamaño, estamos afectando su grado de atención e importancia en relación al resto.
b. Cantidad necesaria: es importante mantener un estándar de componentes, de modo de no confundir al usuario que visita nuestro programa.. Si deseamos implementar alguno que se diferencie con un funcionamiento nuevo, es preciso primero educar a los usuarios sobre el manejo correspondiente.
c. Valores predeterminados: para agilizar la carga de datos regulares, podemos configurar sus valores de manera predeterminada.

Como podemos observar, el hecho de realizar una buena interfaz que se adecue a las necesidades del usuario es una tarea que nos demandará un tiempo considerable.

A continuación, vamos a conocer cómo se trabaja la confección de interfaces gráficas a partir de Visual Basic.

Interfaces de escritorio/ web/móvil

En esta sección vamos a repasar aquellos controles más usuales que utilizaremos en los distintos desarrollos y los compararemos en tres ámbitos posibles: la aplicación de escritorio, las publicaciones web y los dispositivos móviles.

Componentes usuales

Antes detallamos como usuales a los controles etiqueta, caja de texto, casillas de verificación, botón de opción, lista desplegable y grilla de datos. Es importante tener en cuenta que los diferentes controles que vamos a utilizar tienen características particulares, y dependerán del lenguaje de programación, el IDE y la plataforma que empleemos. En los siguientes párrafos veremos una breve presentación de cada control en tres ámbitos diferentes.

> LOS CONTROLES VARÍAN SEGÚN EL LENGUAJE, EL IDE Y LA PLATAFORMA A UTILIZAR

Etiqueta – Label

El control **etiqueta** tiene un uso informativo, ya que se utiliza para presentar información en pantalla. Puede ser para titular la carga de algún control o para dejar avisos en las interfaces. Es posible modificar su contenido para mostrar un resultado, pero en general, se utilizan para etiquetar otros componentes.

A continuación, veamos cómo deben aplicarse las etiquetas en función a los lenguajes de escritorio, Web y móvil.

Escritorio	Web	Móvil
En Visual Basic: Label.Text="ejemplo"	En PHP: <label>ejemplo</label>	En Android: Android:label="ejemplo"

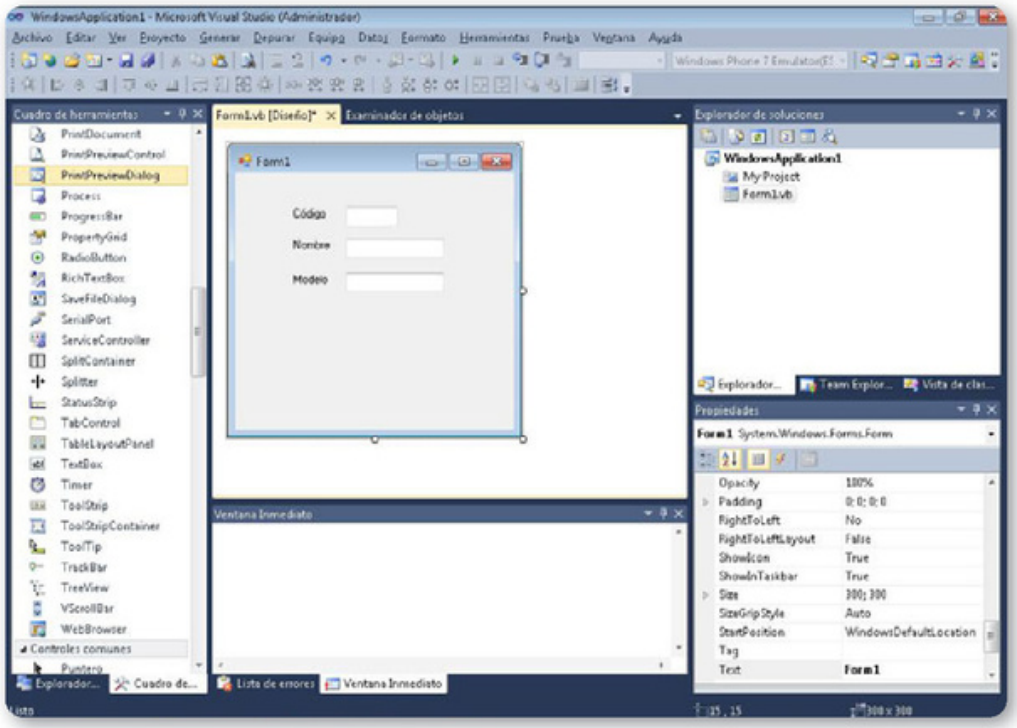


Figura 13. Uso de **`label`** o **`etiqueta`** en diferentes ámbitos y entornos visuales.

Caja de texto – TextBox

La caja de texto se emplea para ingresar valores a una aplicación o solicitar el ingreso de datos a los usuarios. Es el componente más frecuente para datos tales como: nombre, apellido, búsquedas, etc.

En algunos lenguajes de programación, podemos editar varios de sus aspectos, tales como: su tamaño de caracteres, el texto en mayúscula o minúscula, su visualización en forma de contraseña, etc.

A continuación, veamos algunos ejemplos:

CUADRO DE DIÁLOGO

En .NET contamos con varios cuadros de diálogo comunes que se pueden utilizar para mejorar la interfaz de usuario. De esta forma, se obtiene una mayor coherencia a la hora de abrir y guardar archivos, manipular la fuente y el color del texto o, incluso, imprimir.

Escritorio	Web	Móvil
En Visual Basic: TextBox.Text=""	En PHP: <textarea>ejemplo de texto</ textarea> Enlace de ejemplo: www.w3schools.com/tags/ tag_textarea.asp En PHP Form: <input type="text" name="nombre"/>	En Android: Se utiliza un método llamado Input. Enlace de ejemplo: http://android-developers. blogspot.com.ar/2009/04/ creating-input-method.html

Veamos el resultado visual en la **Figura 14**.

Figura 14. Uso de **TextBox/Text/Caja de texto** en diferentes ámbitos y entornos visuales.

GROUPBOX

Los controles **GroupBox** muestran un marco alrededor de un grupo de controles con o sin título. Utilizamos **GroupBox** para agrupar de forma lógica una colección de controles en un formulario. El cuadro de grupo es un control contenedor que puede utilizarse para definir grupos de controles.

Casilla de verificación – CheckBox

Este componente brinda un conjunto de opciones para que el usuario elija. Por lo general, lo encontramos como **CheckBox**, y la propiedad que tiene es chequeado o no chequeado. La falta de marca implica la negación de la afirmación.

Las casillas de verificación funcionan independientemente una de otra. Gracias a este componente, el usuario puede activar varias casillas al mismo tiempo, o ninguna.

Escritorio	Web	Móvil
En Visual Basic: CheckBox.Checked=True	En PHP Form: <input type="checkbox" name="vehicle" value="Bike" /> Una opción Enlace de ejemplo: www.w3schools.com/html/tryit.asp?filename=tryhtml_checkbox	Se utiliza un evento llamado onClick. Enlace de ejemplo: http://developer.android.com/guide/topics/ui/controls/checkbox.html

Veamos el resultado visual:

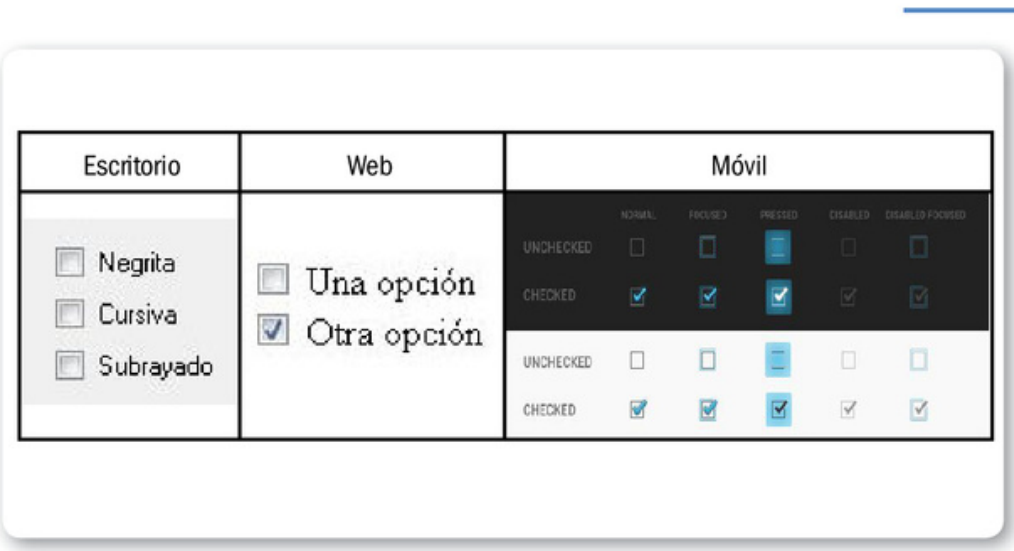


Figura 15. Uso de **CheckBox/Casilla de verificación** en diferentes ámbitos y entornos visuales.

Botón de opción – OptionButton/RadioButton

Con los botones de opción podemos presentar a los usuarios un conjunto de dos o más opciones. Pero, a diferencia de las casillas de verificación, los botones de opción deben funcionar siempre como parte de un grupo; es decir, al activar un botón, los otros no podrán seleccionarse. Adyacente a cada botón de opción, normalmente se muestra un texto que describe la opción que este representa.

Escritorio	Web	Móvil
En Visual Basic: OptionButton.Checked=True	En PHP Form: <input type="radio" name="sexo" value="femenino" /> Femenino Enlace de ejemplo: www.w3schools.com/html/tryit.asp?filename=tryhtml_radio	Se utiliza un RadioButtonGroup y luego la declaración de cada RadioButton. Enlace de ejemplo: http://developer.android.com/guide/topics/ui/controls/radiobutton.html

Veamos el resultado visual:

Figura 16. Utilización de **`OptionButton`/`Botón de opción`** en diferentes ámbitos y entornos visuales.

Lista desplegable – ComboBox/ListBox

Las listas desplegables permiten añadir elementos en forma de lista y también seleccionar elementos de la misma para trabajar los datos. Pueden tener diferentes configuraciones. En algunos casos, podemos establecer valores predeterminados (por ejemplo, el nombre de países para su selección); en otros, podemos ingresar texto, ya sea para buscar dentro de la lista o para agregar un valor a ella.

Escritorio	Web	Móvil
En Visual Basic: ComboBox.Items.Add("Opción 1")	En PHP Form: <select name="select"> <option value="Opción 1" selected>Option 1</option> Enlace de ejemplo: www.abbeyworkshop.com/howto/lamp/php-listbox/index.html	Se puede utilizar un control llamado SPINNER para presentar una lista con opciones. Enlace de ejemplo: http://developer.android.com/guide/topics/ui/controls/spinner.html

Veamos el resultado visual:

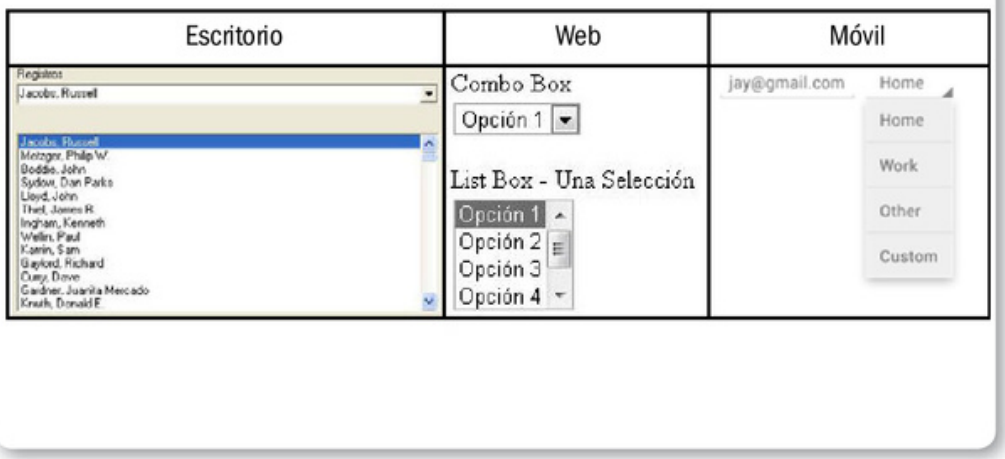


Figura 17. Uso de **`lista desplegable/ListBox/List`** en diferentes ámbitos y entornos visuales.

Grilla de datos – Grid

Los controles de grillas son avanzados, y se utilizan para desplegar información en pantalla. En el caso de algunas interfaces, son utilizados para disponer otros componentes. Tienen varias propiedades y su configuración es amplia.

Escritorio	Web	Móvil
En Visual Basic: Se utiliza un control llamado DataGridView.	En PHP se puede hacer una combinación entre tabla, etiquetas y cajas de texto.	En el caso de Android, GridView es un tipo de visualización en pantalla.

Veamos el resultado visual en la **Figura 18**.

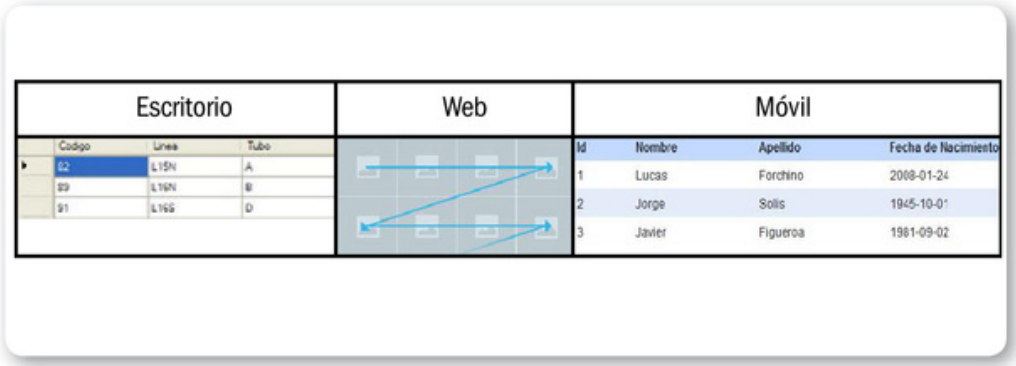


Figura 18. Uso de **`grilla/Grid/GridView`** en diferentes ámbitos y entornos visuales.

Botones de comando

Este componente se usa para darle órdenes particulares al sistema. Esa orden debe estar claramente indicada en el botón, ya sea por su título de texto o por su icono.

Componentes usuales – visuales

Además de los componentes básicos, debemos tener en cuenta otros tipos de objetos que nos servirán para ayudar al usuario con el uso de la aplicación. A continuación, veamos cada uno de ellos.

Iconos

Los iconos son la representación gráfica de diferentes dimensiones, en el rango de 16x16 píxeles a 128x128. Por lo general, caracterizan una acción o un evento que deseamos ejemplificar. Por ejemplo: el icono de imprimir, el de un explorador de Internet, etc. A la hora de seleccionar los iconos que representen las tareas de nuestro desarrollo, debemos asegurarnos de que el usuario los reconozca. Para eso, tendremos que evaluar su impacto a través de un test de usuarios.

Imágenes

Debemos estar atentos en la selección de imágenes, ya que estas deben ayudar a la comprensión y uso de la aplicación. Por eso hay que evitar el uso de gustos particulares, como veíamos en el ejemplo en donde se aplicaba el escudo de un equipo deportivo.

Ventanas

Las aplicaciones de escritorio utilizan ventanas o formas (form) que contienen otros controles. Estas pueden tener diferentes formas, colores y estilos. Por ejemplo, la del Explorador de Windows tiene una característica, y la de Winamp, otra.

Contenedores

Hay componentes que nos permiten contener otros componentes, por ejemplo: las fichas (tabs), cuadros o marcos y cuadros de grupo (groupbox). Se los utiliza para organizar, ordenar o seccionar una tarea con componentes de otras.

PANEL

Panel es un control que contiene otros controles. Se puede utilizar para agrupar colecciones de controles, como un grupo de controles **RadioButton**. Al igual que sucede con otros controles contenedores (como **GroupBox**), si la propiedad **Enabled** del control **Panel** está establecida en **false**, los controles contenidos dentro de **Panel** también se deshabilitarán.

Figura 19. Distintas ventanas de interfaces gráficas en entornos de temáticas diferentes.

Confección de interfaces en Visual Basic

Ahora comencemos a aplicar los conocimientos adquiridos en este capítulo, suponiendo que vamos a generar una aplicación en la que

HERRAMIENTAS EN LA IDE

Actualmente podemos encontrar diferentes IDEs de desarrollo que ofrecen una gran ayuda a la hora de confeccionar las interfaces gráficas. La evolución en el diseño es cada vez más asistido, para evitar errores en la normalización gráfica. De esta forma, contamos con distintas herramientas guías que ayudan en la correcta separación y alineación de los controles, e incluso nos sugieren formas para cargar o mostrar la información.

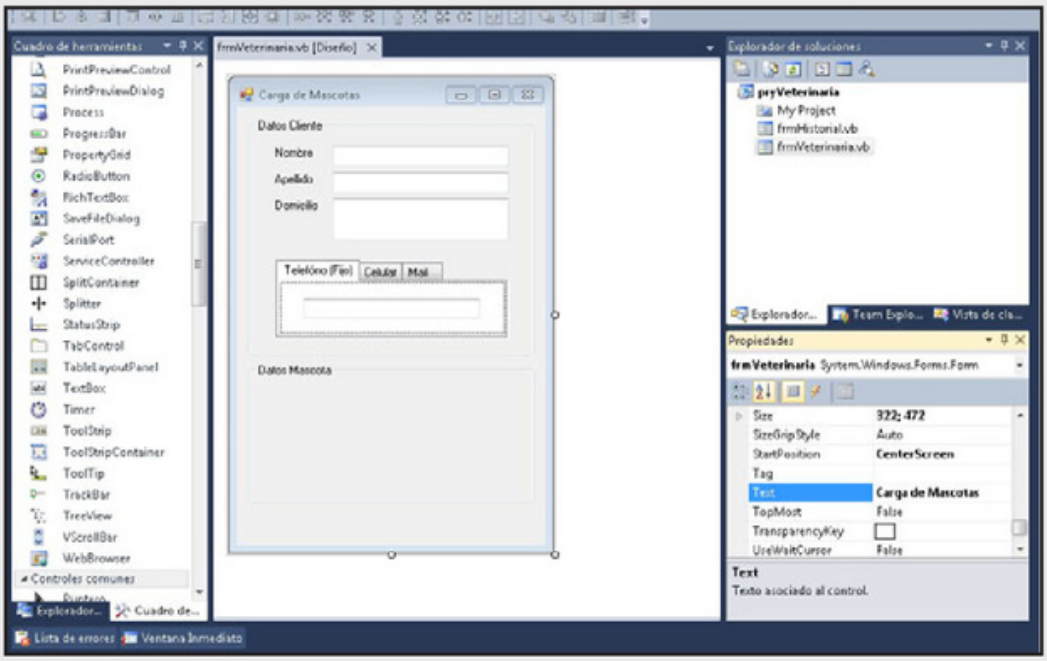
nos encomiendan crear la interfaz gráfica y el comportamiento de un desarrollo. Para eso, nos solicitan lo siguiente:

- **Aplicación de escritorio**: Windows Form.
- **Situación para representar**: el sistema es para una veterinaria, cuya interfaz debe almacenar la siguiente información: nombre de mascota, edad, raza, color, nombre y apellido del dueño, domicilio, teléfono (celular o fijo), vacunas e historial médico.

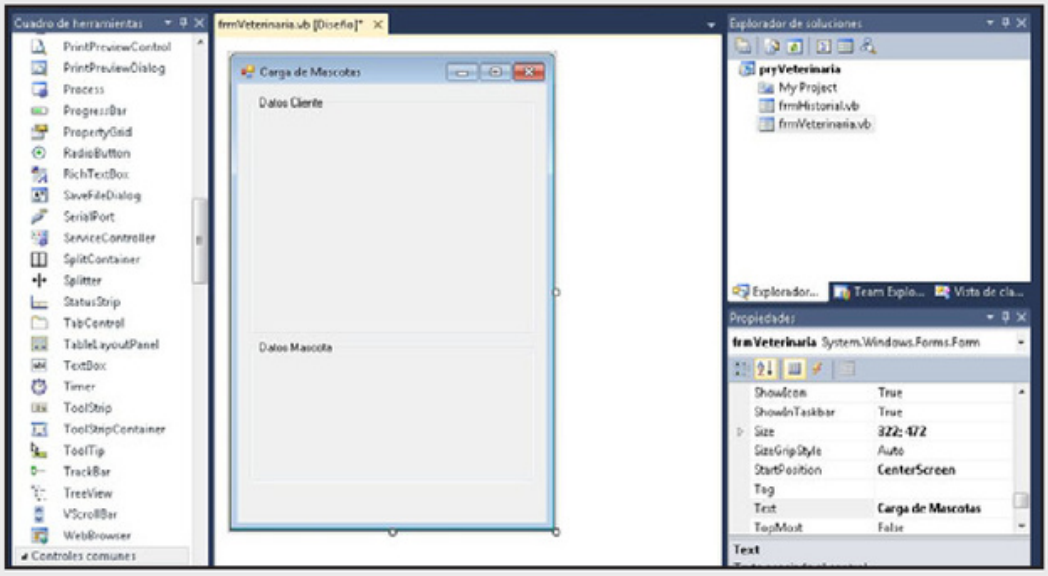
Iniciemos el proyecto como ya hemos practicado anteriormente. Ejecutamos Visual Studio. Luego en el menú **Archivo**, hacemos clic en **Nuevo** y seleccionamos de Visual Basic **Windows Forms**. Grabamos el proyecto como **pryVeterinaria**. Tendremos el formulario por defecto **Form1**, en el cual diseñaremos los controles que sean necesarios para este programa. Veamos el paso a paso de la configuración:

PASO A PASO: CREACIÓN DE PRYVETERINARIA

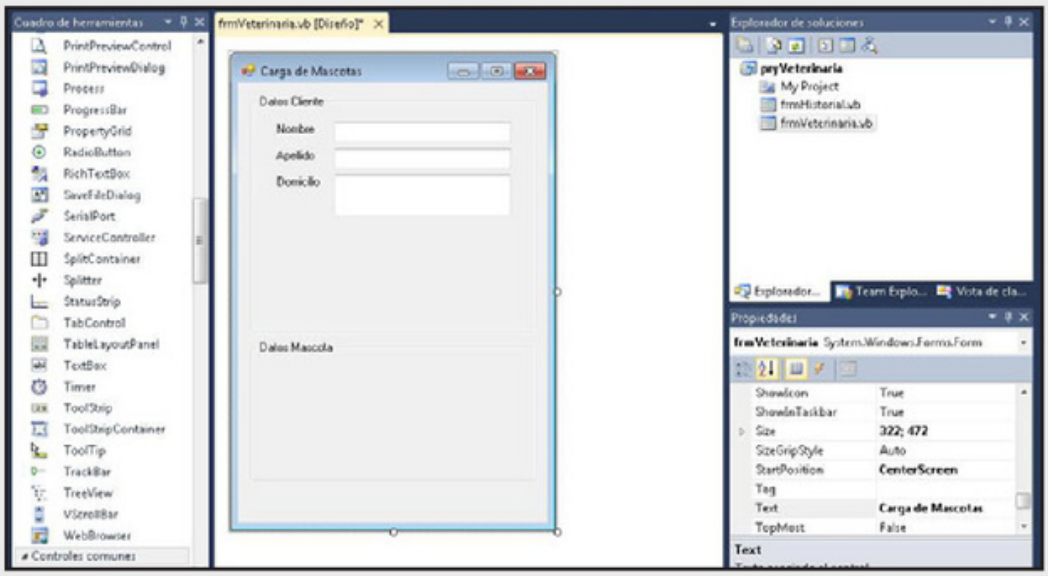
01 Seleccione el **Form1** y asigne las siguientes propiedades: **frmVeterinaria** (en **Name**), **Carga de Mascotas** (en **Text**) y **CenterScreen** (en **StarPosition**).



02 Para ordenar la información que será cargada en la ventana agregue dos **GroupBox** a la interfaz. Al primero asígnele los siguientes valores: **gbDatosCliente** (en **Name**) y **Datos Cliente** (en **Text**). Y al segundo, agregar: **gbDatosMascota** (en **Name**) y **Datos Mascota** (en **Text**).



03 Agregue tres etiquetas y tres cajas de texto a la interfaz.



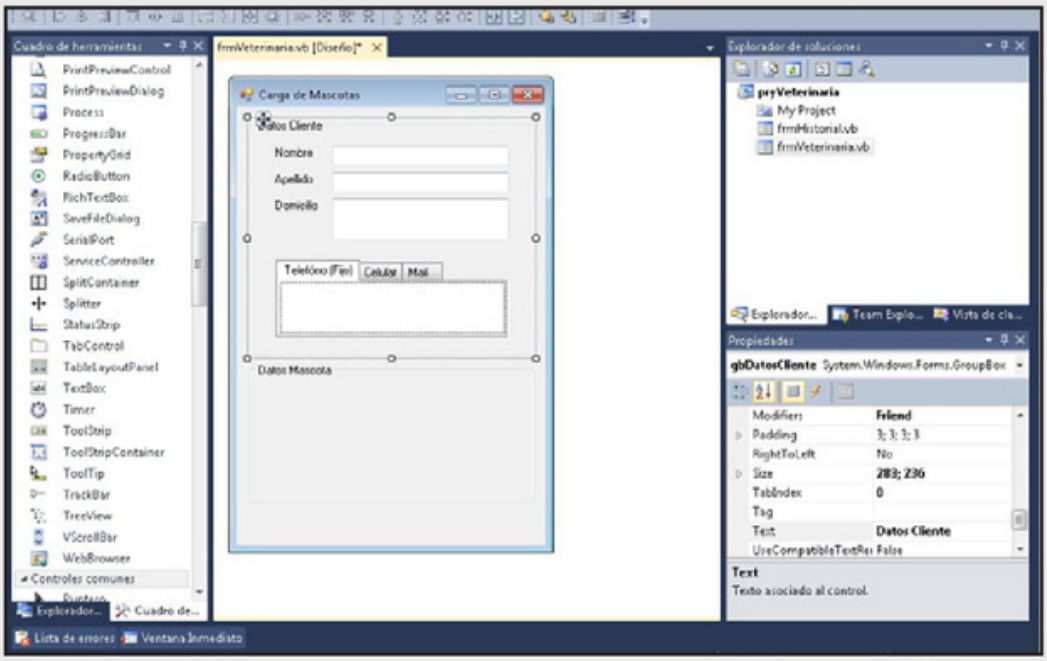
Una vez creado el proyecto y ordenadas las categorías para el ingreso de datos, vamos a confeccionar los controles para almacenar los datos correspondientes a nuestros clientes. Para eso, dentro de **gbDatosCliente** que generamos antes, vamos a asignar los siguientes valores:

- **Para el nombre**: **lblNombre** (en **Label – Name**), **Nombre** (en **Text**), **txtNombre** (en **TextBox – Name**) y 50 (en **MaxLength**).
- **Para el apellido**: **lblApellido** (en **Label – Name**), **Apellido** (en **Text**), **txtApellido** (en **TextBox – Name**) y 50 (en **MaxLength**).
- **Para el domicilio**: **lblDomicilio** (en **Label – Name**), **Domicilio** (en **Text**), **txtDomicilio** (en **TextBox – Name**), 250 (en **MaxLength**), **True** (en **Multiline**) y 175; 41 (en **Size**).

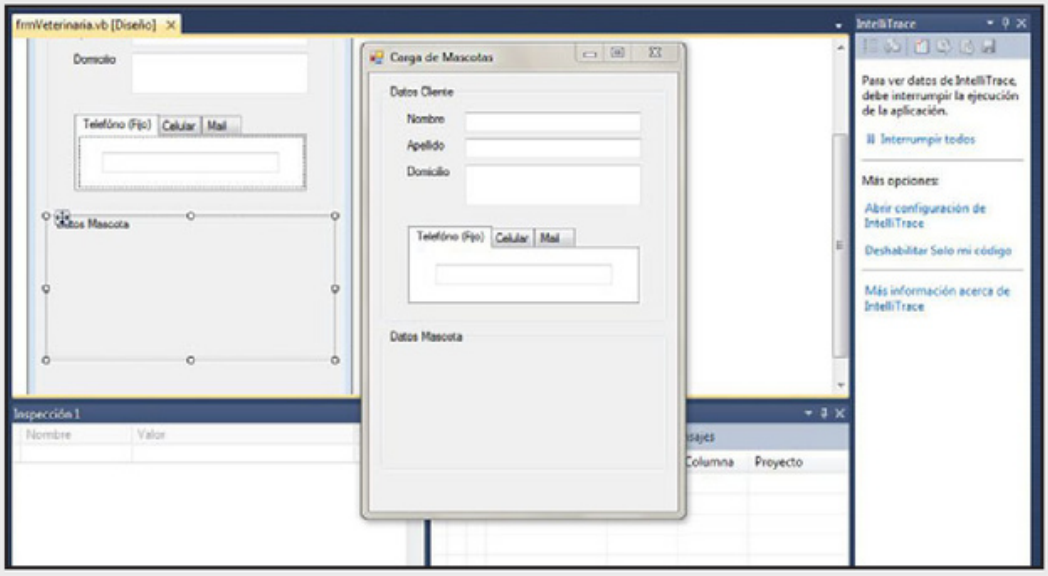
Luego agregaremos un control llamado **Tab Control** para trabajar los contactos referidos al cliente.

PASO A PASO: DATOS DE CLIENTES

01 Asigne **tabContacto** (en **TabControl – Name**). Luego haga clic en **TabPages**: Colecciones... y agregue una **TabPage**. Cambie las propiedades **Text** y **Name**, por: **Telefóno (Fijo)**; **Celular**; **Mail**..



02 Agregue un **TextBox** en cada pestaña para su respectivo contacto. En este caso, utilice un solo ingreso de dato para cada tipo de comunicación, teniendo en cuenta que puede ser más de un mail, celular o teléfono fijo. Luego asigne el nombre de cada **TextBox: txtFijo; txtCelular; txtMail.**



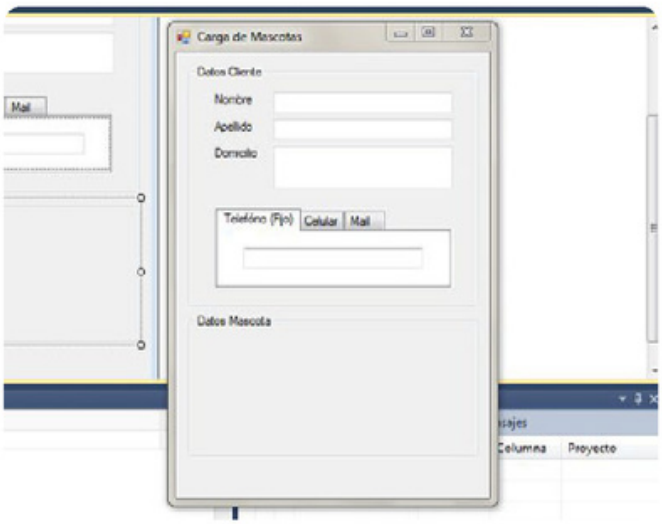


Figura 20. Resultado de la pantalla que diseñaremos sobre Datos de Cliente que serán ingresados.

Siguiendo con la misma lógica aplicada en el ingreso de datos de cliente, confeccionaremos los controles de carga para las mascotas. Para hacerlo, dentro de **gbDatosMascota** que generamos antes, vamos a diseñar lo que vemos en la **Figura 21**.

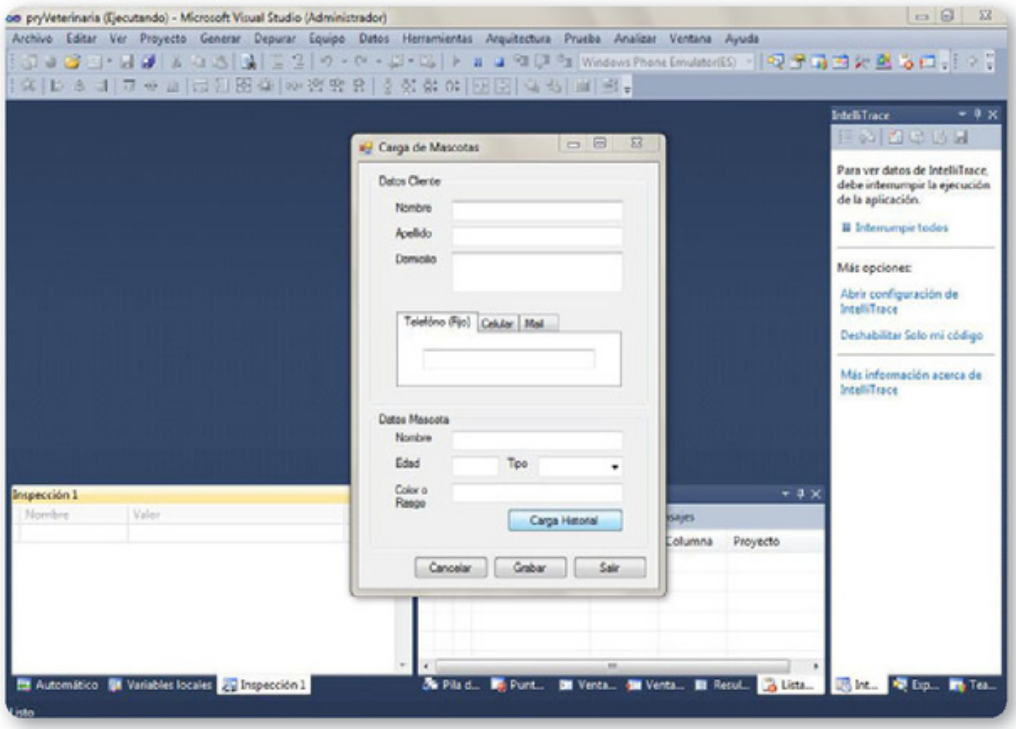


Figura 21. Estos son algunos de los controles que nos serán útiles para la carga de mascota.

Agregaremos cuatro etiquetas, tres cajas de texto, una lista desplegable y un botón de comando. Luego asignaremos:

- Para el nombre de la mascota: **lblNombre** (en **Label** - **Name**), **Nombre** (en **Text**), **txtNombreMascota** (en **TextBox** – **Name**) y **50** (en **MaxLength**).
- Para la edad: **lblEdad** (en **Label** – **Name**), **Edad** (en **Text**), **txtEdad** (en **TextBox** – **Name**) y **2** (en **MaxLength**).
- Para el tipo: **lblTipo** (en **Label** – **Name**), **Tipo** (en **Text**) y **cboTipo** (en **ComboBox** – **Name**).

NECESIDADES DEL CLIENTE

Es importante considerar las necesidades del cliente a la hora de determinar el manejo y la presentación de la Información, pero también podemos sugerir nuevos controles. Recordemos que como programadores podemos desarrollar estructuradamente sobre los controles usuales o fortalecer el diseño y funcionalidad a partir de determinados controles que no se sean utilizados con frecuencia en las aplicaciones habituales.

Para la propiedad correspondiente a **items: colecciones...** agregaremos los siguientes tipos: Canino, Felino.

- Para el color o rasgo: **lblColor** (en **Label – Name**), **Color o Rasgo** (en **Text**), **False** (en **AutoSize**). En este caso, reasignamos el tamaño para que el texto se ajuste al diseño.

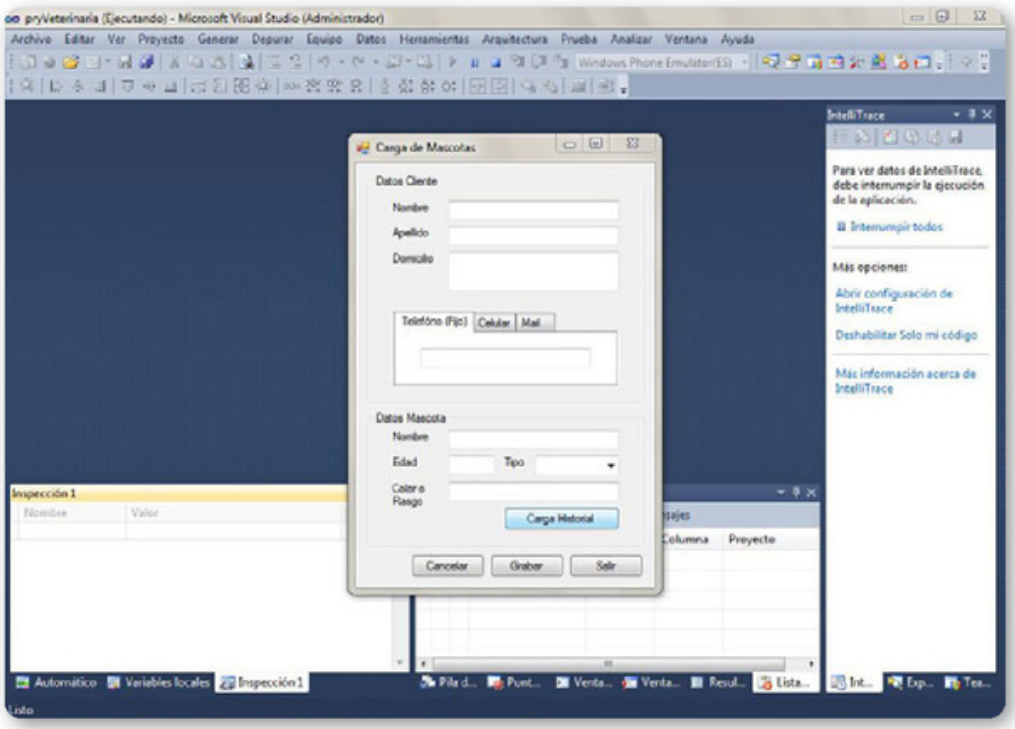


Figura 22. Pantalla de carga completa de información, incluyendo los datos de clientes y mascotas.

Por último, para agregar los botones de comando que se encuentran en la parte inferior de la pantalla, asignaremos:

- Botón **Grabar**: **btnGrabar** (en **Button – Name**) y **Grabar** (en **Text**).
- Botón **Cancelar**: **btnCancelar** (en **Button – Name**) y **Cancelar** (en **Text**).
- Botón **Salir**: **btnSalir** (en **Button – Name**) y **Salir** (en **Text**).

Una vez ingresados todos los datos de las mascotas, vamos a generar un nuevo espacio para cargar información sobre el historial médico. Para eso, agregamos el botón asignando **btnHistorial** (en **Button – Name**) y **Carga Historial** (en **Text**). Luego, vamos a agregar una **WindowsForm** para confeccionar la ventana de carga de datos del

Historial de la mascota y asignar **frmHistorial** (en **Name**). En él vamos a agregar un **GroupBox** y asignarle los controles que podemos observar en la imagen que aparece a continuación.

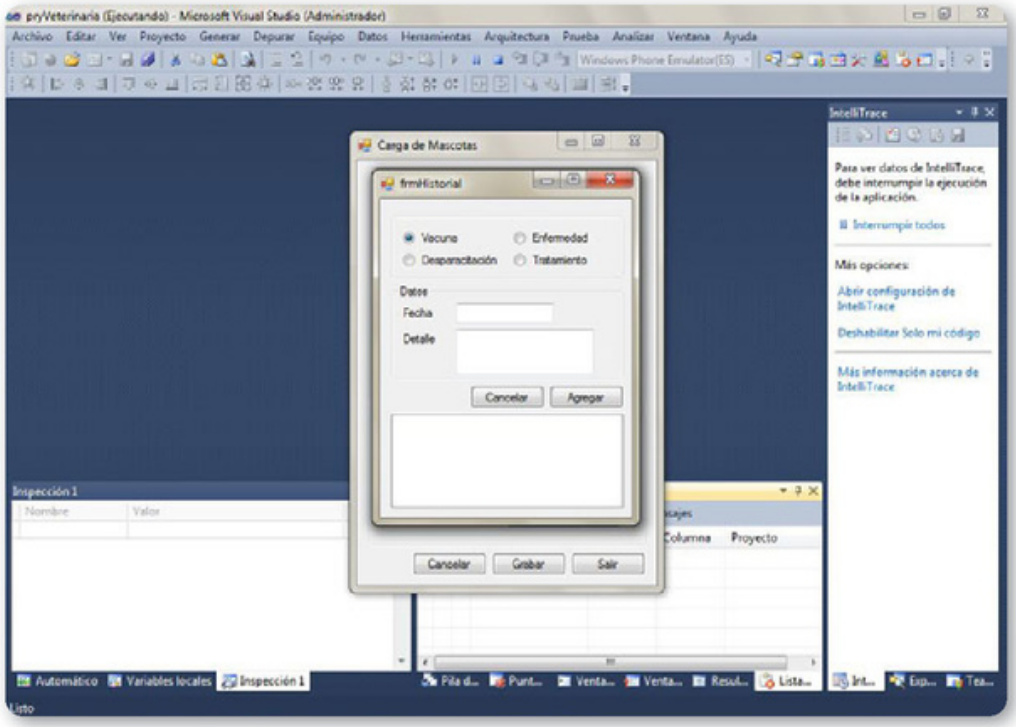


Figura 23. Interfaz gráfica para la pantalla de Historial que nos mostrará en detalle la información cargada.

A continuación, veamos cuál es la asignación correcta que debemos realizar sobre los diferentes controles.

- Para Vacunas: **optVacuna** (en **RadioButton – Name**) y **Vacuna** (en **Text**).
- Para Desparasitación: **optDesparacitacion** (en **RadioButton – Name**) y **Desparasitación** (en **Text**).
- Para Enfermedad: **optEnfermedad** (en **RadioButton – Name**) y **Enfermedad** (en **Text**).
- Para Tratamiento: **optTratamiento** (en **RadioButton – Name**) y **Tratamiento** (en **Text**).

Si deseamos incluir más información que pueda ser útil en el historial, tendremos que agregar **gbDetalle** (en **GroupBox**). Luego dibujaremos los controles y asignaremos lo siguiente:

- Para fecha: **lblFecha** (en **Label** - **Name**) y **Fecha** (en **Text**). En la caja de texto: **txtFecha** (**TextBox** – **Name**).
- Para detalle: **lblDetalle** (en **Label** – **Name**), **Detalle** (en **Label** - **Text**). En la caja de texto: **txtDetalle** (**TextBox** - **Name**) y en la propiedad **Multiline** asignamos **True**.

Por último, para agregar los botones de comando y la lista donde veremos los resultados, realizamos la siguiente configuración:

- Botón **Agregar**: **btnAgregar** (en **Button** – **Name**) y **Agregar** (en **Text**).
- Botón **Cancelar**: **btnCancelar** (en **Button** – **Name**) y **Cancelar** (en **Text**).
- El listado de resultado: **lstHistorial** (**enListBox** – **Name**).

Siguiendo dichas especificaciones, podremos confeccionar una interfaz gráfica que permita cargar toda la información necesaria sobre los clientes y sus mascotas. Recordemos que la funcionalidad de los controles, el orden de la carga, el tamaño de los caracteres y los listados predeterminados siempre van a estar diseñados en función al manejo de los usuarios y sus necesidades.

RESUMEN

En este capítulo hemos recorrido las normas básicas que debemos tener en cuenta al momento de confeccionar una interfaz gráfica, evaluando la importancia de lo visual en la interacción con los usuarios. Luego repasamos los controles más usuales que podemos encontrar en los tres ambientes: web, móvil y de escritorio. Finalmente, confeccionamos una interfaz gráfica de escritorio, en función de un caso práctico de carga de datos. Es importante tener en cuenta que la confección y el diseño de una interfaz gráfica siempre dependerán del modelo de negocio con el que estemos trabajando.

Actividades

TEST DE AUTOEVALUACIÓN

1. ¿Cuáles son las características de las interfaces de escritorio?
2. ¿Cuáles son los controles más frecuentes en la Web?
3. ¿La interacción con el usuario es solo gráfica?
4. ¿Qué es una interfaz gráfica?
5. ¿Cuáles son los controles más frecuentes en móviles?
6. ¿Qué utilidad tienen los **ComboBox**?
7. ¿Qué utilidad tienen las grillas?
8. ¿Cuándo es útil un botón de opción?
9. ¿Cuándo es útil una casilla de verificación?
10. ¿Cuáles son los medios gráficos que podemos utilizar para comunicarnos con el usuario?

*08

Almacenar información en archivos

En casi cualquier desarrollo o aplicación que vayamos a crear, en alguna medida tendremos que realizar el resguardo y el almacenamiento de información. Por eso, en este capítulo veremos algunas opciones útiles y simples para iniciar esta acción con los datos.

- Almacenar en archivo de texto (FileSystem).....368
- Resumen.....375
- Actividades.....376

Almacenar en archivo de texto (FileSystem)

En el caso de Visual Studio, vamos a encontrar herramientas que nos proveen de su framework .NET para manejar archivos de entrada y salida. Existe un espacio de nombre llamado **FileSystem**.

Esta clase se encuentra ubicada por defecto dentro del miembro de **Microsoft.VisualBasic**; por lo tanto, podremos utilizarla desde cualquier proyecto sin necesidad de agregar referencias.

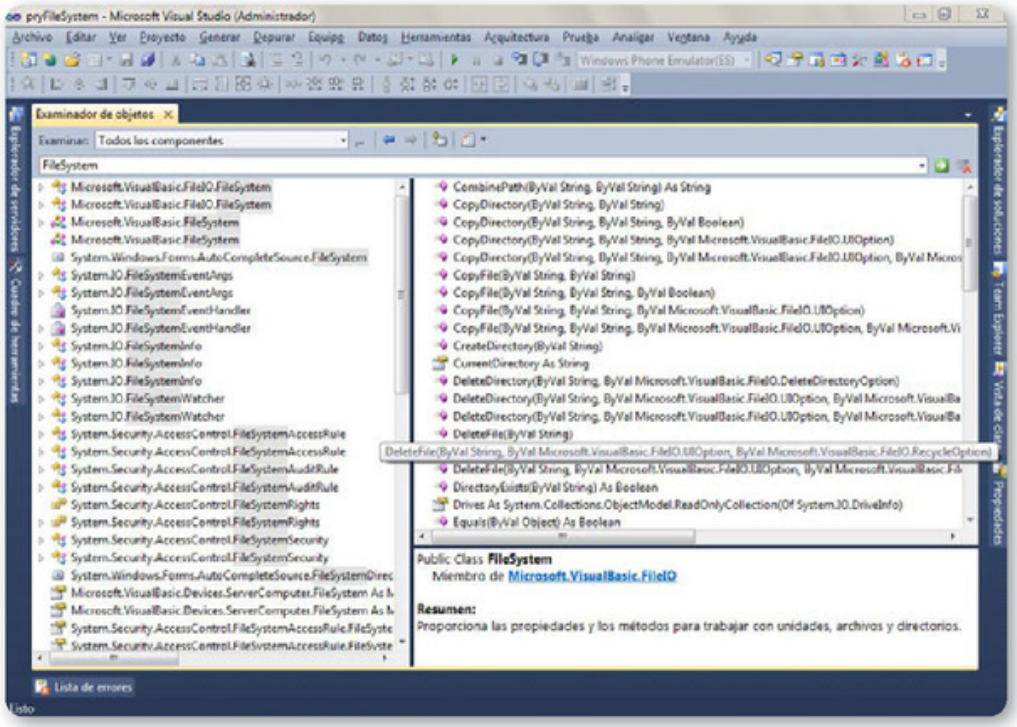


Figura 1. Características de `FileSytem` que encontramos en el examinador de objetos de Visual Studio.

La clase **FileSystem** contiene todos los procedimientos necesarios para realizar operaciones con archivos, carpetas y procedimientos del sistema. Si vamos a grabar información en un archivo de texto, con extensión **TXT**, cabe destacar que con **FileSystem** podemos manipular cualquier tipo de archivo. Ahora vamos a crear un proyecto de Visual Basic y aplicar la práctica de **FileSystem**. Para hacerlo, ejecutamos

Visual Studio, nos dirigimos al menú **Archivo** y hacemos clic en **Nuevo proyecto**, o desde la interfaz principal seleccionamos la misma opción. Luego elegimos **Aplicación de Windows Forms**, hacemos clic en **Aceptar** y grabamos el proyecto como **pryFileSystem**.

Trabajamos sobre el **FORM** y accedemos al **LOAD** del formulario, haciendo doble clic sobre el **FORM**, y aplicamos el siguiente código:

```
FileSystem.FileOpen(1, "ejemploDeArchivo.txt", OpenMode.Append)

FileSystem.Write(1, "Grabando datos en archivo!")
FileSystem.FileClose(1)

MessageBox.Show("Grabe texto :D", "Escribir", MessageBoxButtons.OK)
```

Finalmente, compilamos el proyecto y observamos el resultado que muestra la **Figura 2**.

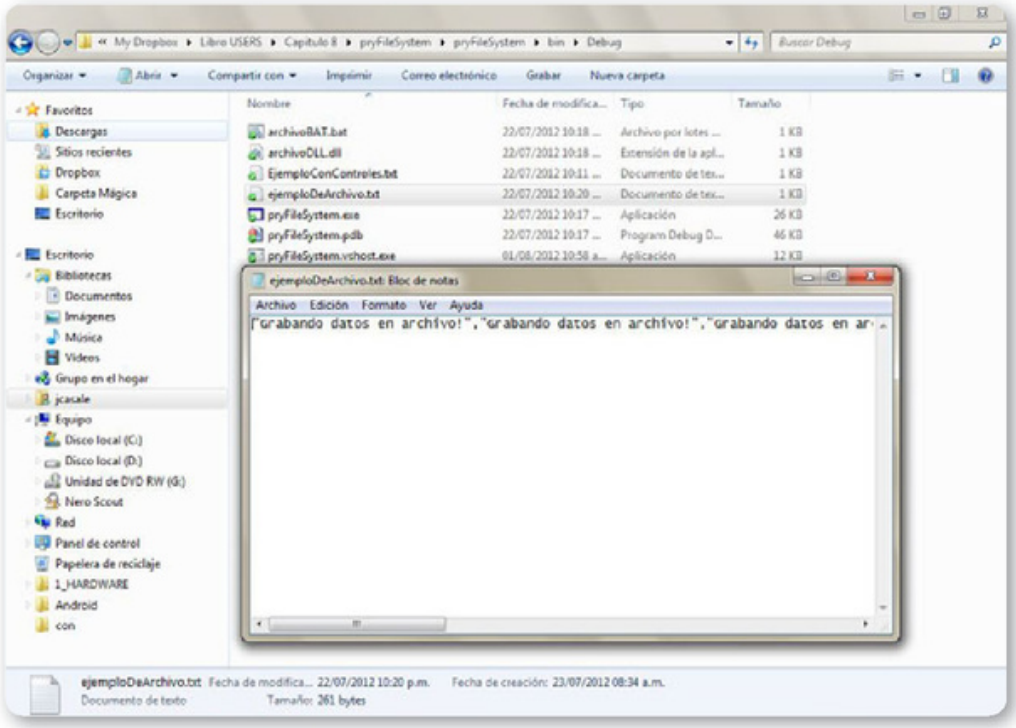


Figura 2. Resultado del ejercicio en un archivo de texto con el nombre **`ejemploDeArchivo.txt`**.

Si buscamos dentro de la carpeta del proyecto en el directorio **BIN**, veremos el archivo de texto que se creó con el proyecto.

Luego analizaremos el código que utilizamos antes, aplicando **FileOpen**, **Write** y **FileClose**. Para seguir avanzando en el tema, veremos en la siguiente tabla el funcionamiento de cada uno de ellos y otras funciones que pueden sernos útiles en la programación.

FUNCIONES

▼ FUNCIÓN	▼ DESCRIPCIÓN	▼ CÓDIGO
FileOpen	Abre un archivo para realizar operaciones de entrada y salida. Para ejecutarla, se deben suministrar tres parámetros obligatorios. Por ejemplo: FileOpen(NúmeroArchivo, NombreArchivo, OpenMode) El primer parámetro es un número para identificar al archivo. El segundo es el nombre del archivo al que podemos especificarle unidad y ruta. El tercero es la enumeración que indica los tipos de operaciones a realizar en el archivo. En el caso de OpenMode define constantes empleadas para establecer los distintos modos de acceso a archivos.	FileSystem.FileOpen(1, "ejemplo-DeArchivo.txt", OpenMode.Append)
OpenMode	Abre los archivos para cumplir distintas funciones: Append: agregar datos Binary: acceso binario Input: lectura Output: escritura Random: acceso aleatorio	Se abre un archivo para lectura: FileOpen(1, "ejemploDeArchivo.TXT", OpenMode.Input) Se abre un archivo para escritura: FileOpen(1, "ejemploDeArchivo.TXT", OpenMode.Output)
FileClose	Concluye las operaciones de entrada y salida de un archivo abierto, y recibe como parámetro el número de archivo a cerrar. Para cerrar todos los archivos abiertos con FileOpen existe la posibilidad de ejecutar una función llamada Reset, que no necesita parámetros.	FileOpen(1, "ejemplo1.TXT", OpenMode.Output) FileOpen(2, "ejemplo2.TXT", OpenMode.Output) FileClose(2) FileClose(1)

Write	Graba datos en un archivo secuencial que se leen con la función Input. Tiene dos parámetros: el primero es el número de archivo, y el segundo es una o más expresiones delimitadas por una coma, que van a ser grabadas en el archivo.	FileSystem.Write(1, "Grabando datos en archivo!")
WriteLine	Trabaja igual que la función Write, pero la diferencia es que graba al final de la salida un carácter de salto de línea (salto de carro y nueva línea o los caracteres chr(13) y chr(10)).	WriteLine(1, "No es necesario decir todo lo que se piensa, lo que sí es necesario es pensar todo lo que se dice.")
Input	Lee datos de un archivo secuencial abierto y les asigna una variable. La función recibe dos parámetros: el primero es el número de archivo, y el segundo es la variable en la que se almacenan los datos.	Dim varTexto As String FileOpen(1, "ejemploDeArchivo.txt", FileMode.Input) Input(1, varTexto) FileClose(1)
LineInput	Se utiliza para leer una línea completa de un archivo secuencial abierto. Los datos de un archivo se leen si fueron escritos con la función Print. La función LineInput lee un conjunto de caracteres y los almacena en una variable. La lectura se detiene cuando se encuentra un retorno de carro o un retorno de carro y salto de línea.	Dim varTexto As String FileOpen(1, "ejemploDeArchivo.txt", FileMode.Input) LineInput(1, varTexto) FileClose(1)

Tabla 1: Conjunto de funciones que podemos aplicar en la programación.

Tomando estas funciones, ahora vamos a retomar el proyecto de **FileSystem** que veníamos trabajando en el paso a paso anterior. Para hacerlo, vamos a agregar una caja de texto y un botón de comando llamado **Grabar desde caja de texto**, aplicando el siguiente código:

FILESYSTEM – VISUAL BASIC

El módulo **FileSystem** contiene los procedimientos empleados para realizar operaciones de archivo, de directorio o carpeta y de sistema. La característica **My** proporciona mayor productividad y rendimiento en las operaciones de entrada y salida de archivos que si se utilizara el módulo **FileSystem**.

```
FileOpen(1, "EjemploConControles.txt", OpenMode.Append)
Write(1, TextBox1.Text)
FileClose(1)
MessageBox.Show("Grabe texto :D", "Escribir", MessageBoxButtons.OK)
```

Luego debemos compilar y comprobar los resultados. Es importante no olvidarnos de revisar la carpeta **BIN** del proyecto.

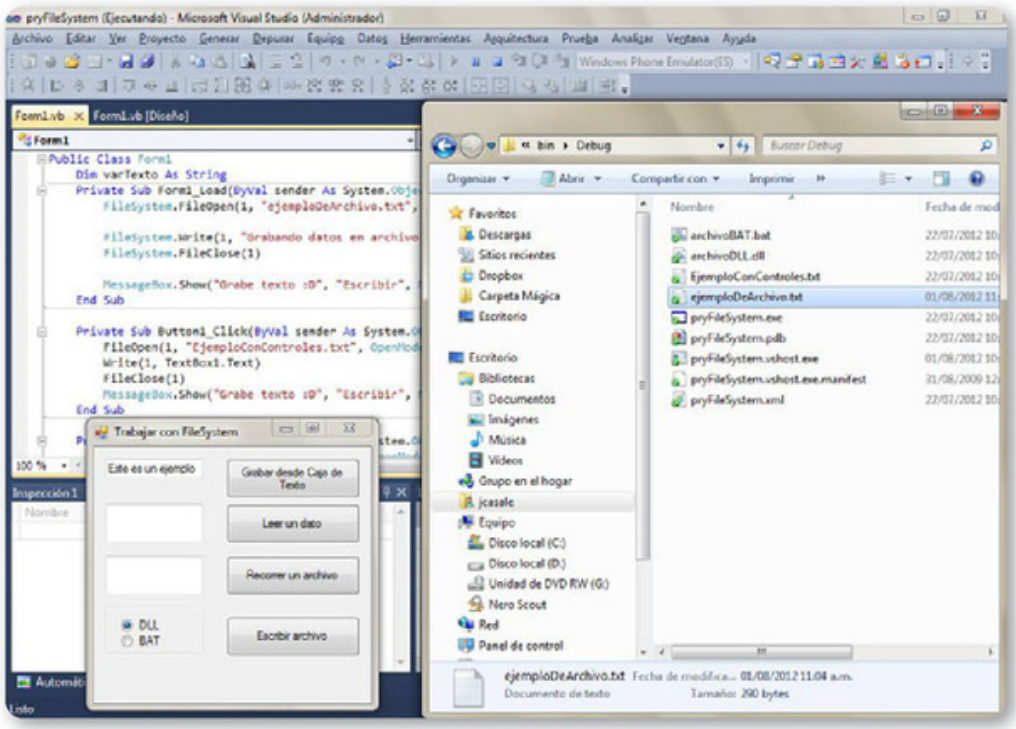


Figura 3. Resultado del ejercicio en donde podemos observar el código de fondo, la aplicación y los archivos generados.

Para continuar, agregaremos otra caja de texto y un botón de comando llamado **Leer un dato**. Dentro de él aplicaremos el siguiente código:

```
FileOpen(1, "EjemploConControles.txt", OpenMode.Input)

Input(1, varTexto)

TextBox2.Text = varTexto
```

```
FileClose(1)
```

Una vez ingresado el código, debemos compilar y comprobar los resultados en la interfaz gráfica. Luego agregaremos otra caja de texto y un botón de comando llamado **Recorrer un archivo**, en donde aplicaremos el siguiente código:

```
FileSystem.FileOpen(1, "EjemploConControles.txt", OpenMode.Input)

    Do While Not EOF(1)
      FileSystem.Input(1, varTexto)

      TextBox3.Text = varTexto
    Loop

    FileSystem.FileClose(1)
```

Es fundamental tener en cuenta que este mismo código nos será útil cuando tengamos un archivo con saltos de líneas.

Para concluir con nuestro proyecto, agregaremos dos botones de opción y un botón de comando que diga **Escribir archivo**, en donde aplicaremos el siguiente código:

```
If optBAT.Checked = True Then
      FileOpen(1, "archivoDLL.dll", OpenMode.Append)
      Write(1, "Creo un archivo de sistema")
      FileClose(1)
      MessageBox.Show("Grabe :D", "Escribir", MessageBoxButtons.OK)
    Else
      FileOpen(1, "archivoBAT.bat", OpenMode.Append)
      Write(1, "Creo un archivo de sistema")
      FileClose(1)
      MessageBox.Show("Grabe :D", "Escribir", MessageBoxButtons.OK)
    End If
```

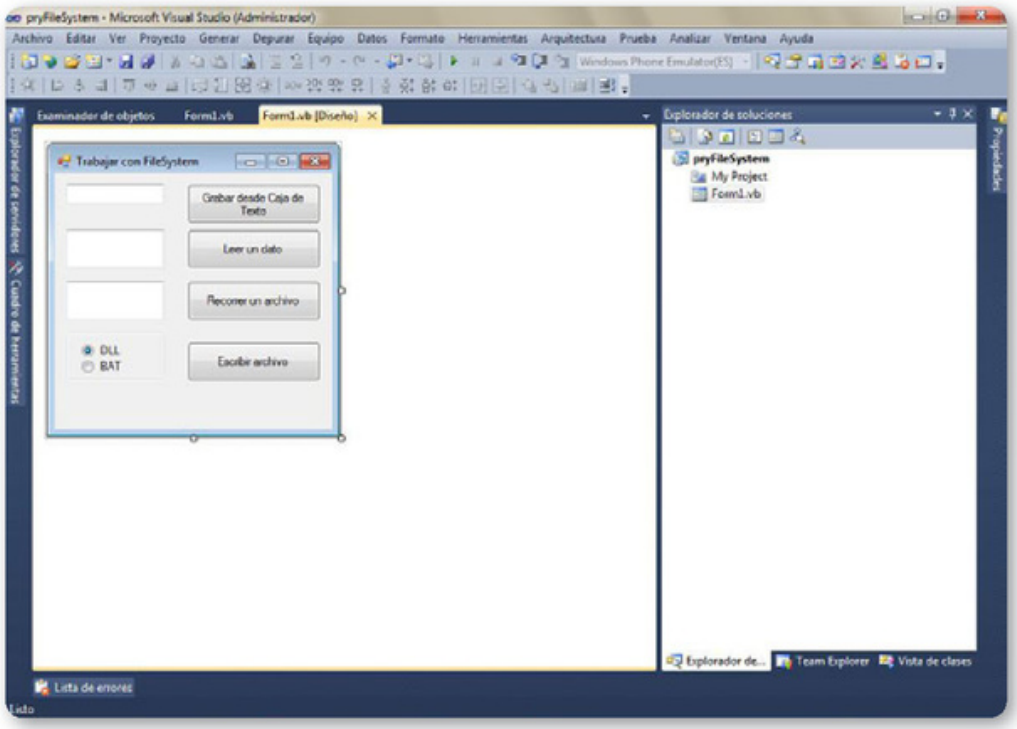


Figura 4. Interfaz gráfica completa con todos los controles necesarios para la ejercitación.

Como podemos observar, en el ejemplo anterior de código hemos logrado grabar, leer e incluso reemplazar distintos archivos con las líneas provistas por **FileSystem** que nos permitieron almacenar información en este primer paso.

El siguiente paso que debemos seguir en la programación es el almacenamiento en base de datos que podemos encontrar en títulos específicos de algún lenguaje de programación.

Para obtener información más detallada sobre el resguardo y el almacenamiento de datos en el sistema, podemos ingresar al sitio web

ARCHIVO - EXTENSIÓN

Cuando vemos **.EXE**, **.DLL**, **.BAT**, etc., debemos saber que se refiere a la extensión de un archivo. Esta es una manera de reconocer el tipo de programa que se necesita para abrir el archivo. No ofrece certidumbre respecto al tipo de archivo, ya que cuando se modifica la extensión, el tipo de archivo no cambia.

de la editorial **www.redusers.com**, o también consultar otro de los libros publicados por la editorial: *Acceso de datos con ADO.NET*, escrito por Fernando Riberi.

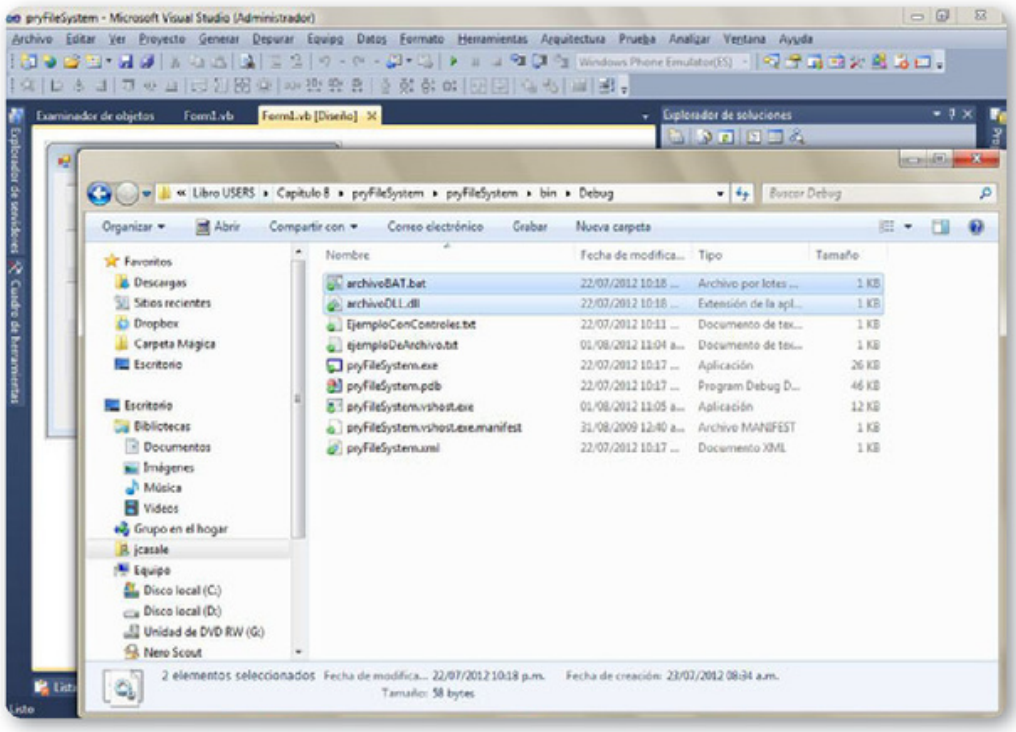


Figura 5. Creación de archivos de diferentes extensiones.

RESUMEN

En este capítulo, hemos conocido algunas opciones simples y útiles para iniciar el almacenamiento de información en archivos. A lo largo de nuestra práctica en Visual Basic, aprendimos cómo se realiza el manejo básico para un archivo de texto y vimos que podemos manipular diferentes tipos de archivos. Ahora que tenemos en claro estas herramientas, podemos encarar pequeñas aplicaciones que requieran almacenar cualquier tipo de información.

Actividades

TEST DE AUTOEVALUACIÓN

1	¿Qué es **FileSystem**?
2	¿Qué es un archivo?
3	¿Qué formas de apertura de archivo existen?
4	¿Qué puede hacer con **FileOpen**?
5	¿Qué características tiene **WriteLine**?
6	¿Qué características tiene **Write**?
7	¿Cómo funcionan **Open** y **Close**?
8	¿Es posible abrir cualquier tipo de archivo?
9	¿Es posible crear cualquier tipo de archivo?
10	¿Qué es un **Input**?

Servicios al lector

En esta sección nos encargaremos de presentar un útil índice temático para que podamos encontrar en forma sencilla los términos que necesitamos.

▾ Índice temático.....378

Índice temático

A

Actores 59/60
Álgebra de Boole 84
Algoritmo 27/107
Almacenamiento de información..... 172
Ámbito de las variables 121/207
Amiga..... 343
Análisis de sistema..... 51
Aplicaciones de escritorio..... 19
Aplicaciones móviles 20
Aplicaciones web 18
Aritméticos 81/154/214
Arrays..... 243/246/248
Asignación 88/213/215
Asignación compuesta 215
Asignación de valores..... 88
Asignación destructiva 91
Aumentar y disminuir 216

B

Beneficios del pseudocódigo 93
Botón de opción 353
Botones de comando 355/363/365
Brainstorming 56

C

Caja de texto 350
Capacitación y formación del usuario..... 66
Casilla de verificación 135/138/352
Caso de uso..... 59/60
Ciclo de vida 41/42/44/47
Clasificación de las variables 77
CLI..... 137
Código 34/148
Cola..... 323
Cómo se utilizan
los operadores..... .80/154/213
Compilador 34/134

C

Componentes usuales 349/355
Comportamiento de la interfaz 178
Conceptos básicos del código 199
Confección de prototipos 64
Constantes declarados 212
Constantes definidas..... 212
Constantes literales..... 211
Creación de proyectos 140
Cuestionarios 56

D

Datos estructurados..... 109/168/253
Datos simples..... 75/95/252
Declaración de variables 204
Desarrollo convencional 50
Desarrollo de aplicaciones..... 14
Desarrollo estructurado..... 50
Desarrollo orientado a objetos..... 51
Diagnóstico 52/53
Diagrama de casos de uso..... 58
Diagrama de flujo..... 27/32/72
Diagrama N-S..... 73
Diseño de un sistema..... 34/57
Diseño de interfaz 336/345

E

Entrada y salida de información 92/239
Espacios de nombre 196/198
Estructura secuencial 95/72
Estructuras condicionales 95/159/230
Estructuras contiguas 253
Estructuras de control 94/159/230
Estructuras dinámicas
y estáticas..... 256/257/273
Estructuras dinámicas
y punteros 254/257/259
Estructuras dobles..... 97/161

E

Estructuras enlazadas 254
Estructuras múltiples 100/162
Estructuras repetitivas 104/165/167/234
Estructuras selectivas..... 232
Estructuras simples 94/95/160/253
Estudio de documentación..... 56
Etapas en la resolución
de un problema 34/117
Etiqueta 138/187/349

F

Fases del desarrollo 41/42/44
For/Loop..... 235
Formas de declarar constantes 211
Framework 38/190
Freeware 18/19/23/38
Función Exit 242
Funciones 42/120/121/370
Funciones del ciclo de vida..... 42/44

G

Generalidades sobre metodologías..... 50
Grilla de datos 355
GUI 134

I

IDE 142/143/194
IDE Sharp Develop 190/193/201
Identificadores..... 80/146
Implementación del desarrollo 65
Inicializar variables..... 207
Interactuar con el usuario 224
Interfaces de usuario..... 337
Interfaces en Visual Basic 357
Interfaces gráficas 134/135/336/357
Interpretación de aplicaciones..... 20

L

Lenguaje Basic 17
Lenguaje C 21/195
Lenguaje de alto nivel 29/133
Lenguaje de aplicaciones..... 29

L

Lenguaje de bajo nivel 29/132/133
Lenguaje de medio nivel..... 29/133
Lenguaje de programación 28/132/140
Lenguaje de redes..... 133
Lenguaje máquina..... 22/29/70/132
Lenguaje objeto 133
Librerías..... 159/247
Límite del sistema..... 59/60
Lista..... 110/268
Listas doblemente enlazadas 293/308
Listas enlazadas..... 269/271
Los sentidos..... 23

M

Manejo de datos en C++ 203
Matriz 114/168
Messagebox 150/154
Metodologías ágiles 39/51
Metodologías de desarrollo 38/39/67
Metodologías iterativas..... 39
Metodologías pesadas 39/45/48
Metodologías tradicionales..... 39/40
Modelo de desarrollo incremental 47/48
Modelo de prototipos 48/49
Modelo en cascada..... 44/45
Modelo en espiral 49
Modelo en V..... 45/46
Modelo iterativo 46/47
Modelos de ciclo de vida 44
Multiplataforma..... 193
MVA..... 145

N

Nodos 255/259/261
Nomenclatura 138/139
Normas de diseño de interfaz 336
Normas de escritura..... 79
Normas ISO..... 43
Normas para el pseudocódigo..... 71
Nuevos dispositivos 14

O

Objetivo de los prototipos 63
Observación 55
Olfato 23
Open Source 58
Operador coma..... 220/236
Operador condicional..... 219
Operador sizeof..... 222
Operadores aritméticos 154/155/214
Operadores bitwise..... 220/221/223
Operadores de asignación..... 213/215/217
Operadores lógicos 82/156/217
Operadores relacionales
y de igualdad..... 157/216
Orden en la programación 94/159

P

Pila 312
Portal de bancos 19
Portal de juegos..... 19
Portal educativo..... 19
Precedencia de los operadores 222
Primera aplicación en C++..... 201
Primeros pasos en el código 148
Procedimientos..... 127
Programación lógica 15
Propiedad Enabled..... 181/182
Propiedades de un FORM..... 177
Propósitos para aprender a desarrollar.... 15
Prototipo desechable..... 63
Prototipo evolutivo..... 63
Prototipos..... 62
Pseudocódigo..... 34/71
Punteros..... 257/259/263

R

Recolectar información 54
Recorrido de información 174
Relacionales 59/86
Relaciones 59/60/67
Relevamiento..... 52/53

R

Resolver problemas..... 16/34/117
Retroalimentación 26
Roles profesionales 43
RUP 40/45

S

Salto de declaraciones 239
Sentencia break..... 239
Sentencia continue..... 241
Sentencia goto..... 241/242
Sistemas operativos 345
Software libre..... 18
Subalgorítmos..... 120

T

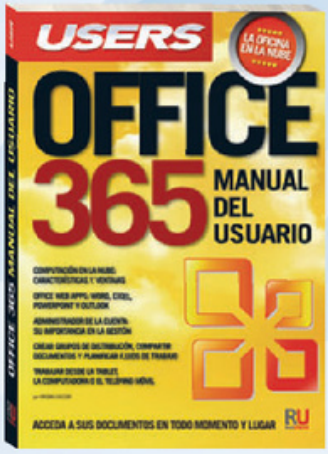
Tareas de un desarrollador 26
Teclas rápidas 160
Tecnología Web..... 40
Test de ejecución..... 34
Testing..... 52/65
Tiempos en la programación..... 147
Tipos de aplicaciones 18
Tipos de datos..... 74
Tipos de datos estructurados..... 109
Tipos de estructuras 252
Tipos de lenguajes..... 132/133
Tipos de metodologías..... 39
Tipos de variables 78
Tormenta de ideas..... 56
Trabajo Freelance 16/ 17
Trial..... 19

U

UML..... 46/57
Uso de controles básicos..... 175
Utilizar funciones y procedimientos 120

V

Variables 75/146
Vector..... 110
Vista..... 23
Visual Basic..... 140

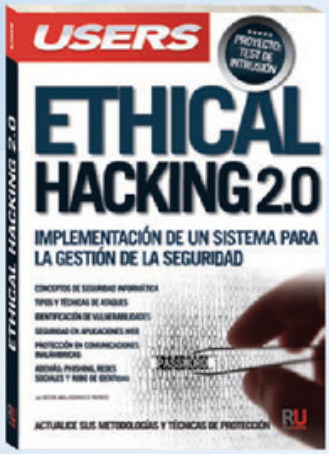


Una obra ideal para aprender todas las ventajas y servicios integrados que ofrece Office 365 para optimizar nuestro trabajo.

→ 320 páginas / ISBN 978-987-1857-65-4

Esta obra presenta las mejores aplicaciones y servicios en línea para aprovechar al máximo su PC y dispositivos multimedia.

→ 320 páginas / ISBN 978-987-1857-61-6



Esta obra va dirigida a todos aquellos que quieran conocer o profundizar sobre las técnicas y herramientas de los hackers.

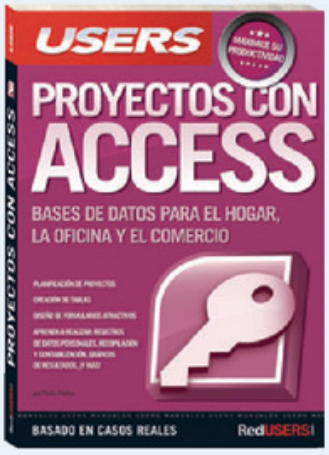
→ 320 páginas / ISBN 978-987-1857-63-0

Este libro se dirige a fotógrafos amateurs, aficionados y a todos aquellos que quieran perfeccionarse en la fotografía digital.

→ 320 páginas / ISBN 978-987-1857-48-7

En este libro encontraremos una completa guía aplicada a la instalación y configuración de redes pequeñas y medianas.

→ 320 páginas / ISBN 978-987-1857-46-3



Esta obra está dirigida a todos aquellos que buscan ampliar sus conocimientos sobre Access mediante la práctica cotidiana.

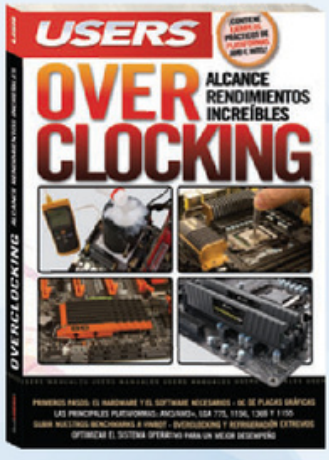
→ 320 páginas / ISBN 978-987-1857-45-6

Este libro nos introduce en el apasionante mundo del diseño y desarrollo web con Flash y AS3.

→ 320 páginas / ISBN 978-987-1857-40-1

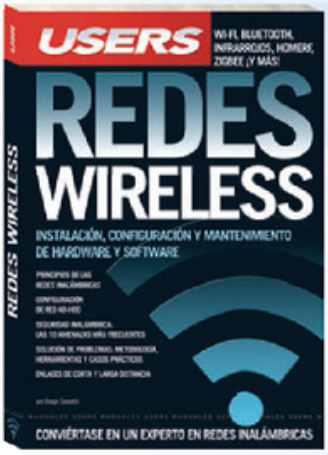
Esta obra presenta un completo recorrido a través de los principales conceptos sobre las TICs y su aplicación en la actividad diaria.

→ 320 páginas / ISBN 978-987-1857-41-8



Este libro está dirigido tanto a los que se inician con el overclocking, como a aquellos que buscan ampliar sus experiencias.

→ 320 páginas / ISBN 978-987-1857-30-2



Este manual único nos introduce en el fascinante y complejo mundo de las redes inalámbricas.

→ 320 páginas / ISBN 978-987-1773-98-5

Esta increíble obra está dirigida a los entusiastas de la tecnología que quieran aprender los mejores trucos de los expertos.

→ 320 páginas / ISBN 978-987-1857-01-2

Esta obra se encuentra destinada a todos los desarrolladores que necesitan avanzar en el uso de la plataforma Adobe Flash.

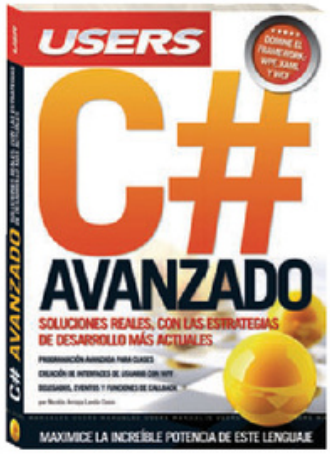
→ 320 páginas / ISBN 978-987-1857-00-5

Un libro clave para adquirir las herramientas y técnicas necesarias para crear un sitio sin conocimientos previos.

→ 320 páginas / ISBN 978-987-1773-99-2

Una obra para aprender a programar en Java y así insertarse en el creciente mercado laboral del desarrollo de software.

→ 352 páginas / ISBN 978-987-1773-97-8



Este libro presenta un nuevo recorrido por el máximo nivel de C# con el objetivo de lograr un desarrollo más eficiente.

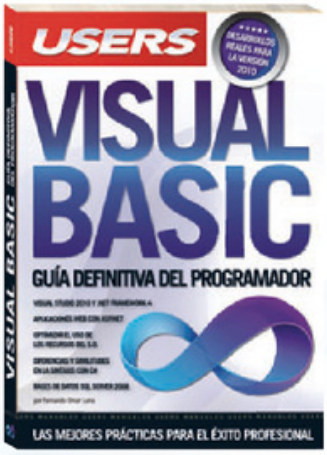
→ 320 páginas / ISBN 978-987-1773-96-1

Esta obra presenta todos los fundamentos y las prácticas necesarios para montar redes en pequeñas y medianas empresas.

→ 320 páginas / ISBN 978-987-1773-80-0

Una obra única para aprender sobre el nuevo estándar y cómo aplicarlo a nuestros proyectos.

→ 320 páginas / ISBN 978-987-1773-79-4



Un libro imprescindible para aprender cómo programar en VB.NET y así lograr el éxito profesional.

→ 352 páginas / ISBN 978-987-1773-57-2

Una obra para aprender los fundamentos de los microcontroladores y llevar adelante proyectos propios.

→ 320 páginas / ISBN 978-987-1773-56-5

Un manual único para aprender a desarrollar aplicaciones de escritorio y para la Web con la última versión de C#.

→ 352 páginas / ISBN 978-987-1773-26-8

Un manual imperdible para aprender a utilizar Photoshop desde la teoría hasta las técnicas avanzadas.

→ 320 páginas / ISBN 978-987-1773-25-1

Una obra imprescindible para quienes quieran conseguir un nuevo nivel de profesionalismo en sus blogs.

→ 352 páginas / ISBN 978-987-1773-18-3

Un libro único para ingresar en el apasionante mundo de la administración y virtualización de servidores.

→ 352 páginas / ISBN 978-987-1773-19-0

Esta obra permite sacar el máximo provecho de Windows 7, las redes sociales y los dispositivos ultraportátiles del momento.

→ 352 páginas / ISBN 978-987-1773-17-6

Este libro presenta la fusión de las dos herramientas más populares en el desarrollo de aplicaciones web: PHP y MySQL.

→ 432 páginas / ISBN 978-987-1773-16-9

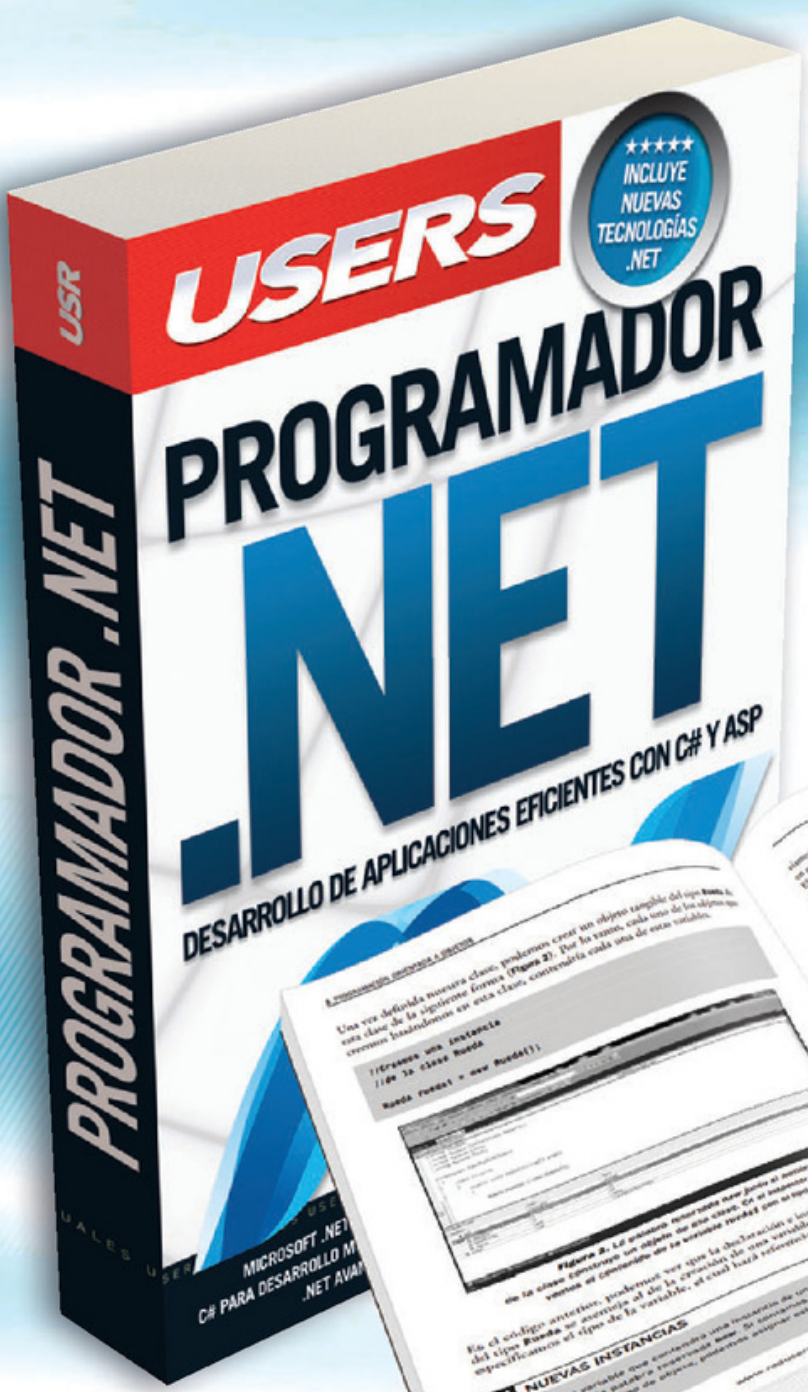
Este manual va dirigido tanto a principiantes como a usuarios que quieran conocer las nuevas herramientas de Excel 2010.

→ 352 páginas / ISBN 978-987-1773-15-2

Esta guía enseña cómo realizar un correcto diagnóstico y determinar la solución para los problemas de hardware de la PC.

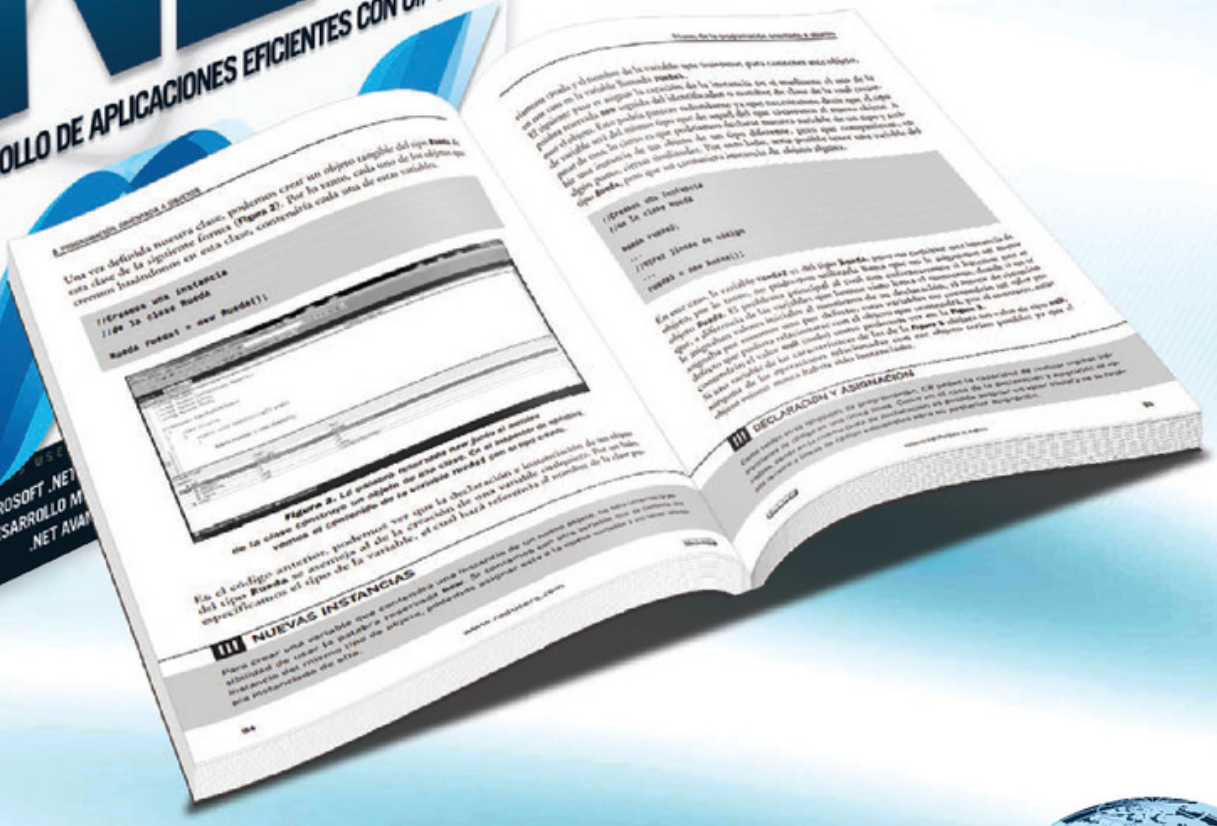
→ 320 páginas / ISBN 978-987-1773-14-5

CONÉCTESE CON LOS MEJORES LIBROS DE COMPUTACIÓN



Este libro está dirigido a todos aquellos que quieran iniciarse en el desarrollo bajo lenguajes Microsoft. A través de los capítulos del manual, aprenderemos sobre POO, la programación con tecnologías .NET y de qué manera se desenvuelven con otras tecnologías existentes.

» DESARROLLO
» 352 PÁGINAS
» ISBN 978-987-1773-26-8



LLEGAMOS A TODO EL MUNDO VÍA OCA* Y DHL**

MÁS INFORMACIÓN / CONTÁCTENOS

usershop.redusers.com +54 (011) 4110-8700 usershop@redusers.com

* SÓLO VÁLIDO EN LA REPÚBLICA ARGENTINA // ** VÁLIDO EN TODO EL MUNDO EXCEPTO ARGENTINA

USERS

INTRODUCCIÓN A LA PROGRAMACIÓN

Presentamos un libro ideal para todos aquellos que quieran iniciarse en el mundo de la programación y conocer las bases necesarias para generar su primer software. A lo largo de su contenido, analizaremos el contexto que impulsa el desarrollo de aplicaciones y las partes que lo constituyen. Una vez adquiridos estos conocimientos, aprenderemos la importancia del pseudocódigo, que nos permitirá trabajar con cualquier lenguaje de programación. A continuación, repasaremos la estructuración de datos para entender cómo funciona su lógica y, así, armar los prototipos de aplicaciones, ya sean de escritorio, web o móviles.

A través de explicaciones sencillas, guías visuales y procedimientos paso a paso, el lector descubrirá una obra imperdible para adquirir bases sólidas en el desarrollo de aplicaciones y aprender a programar de manera eficiente.

> **Conociendo el manejo y la confección de los programas, podremos comprender la lógica propia de la programación y trabajar en cualquier tipo de lenguaje.**

EN ESTE LIBRO APRENDERÁ:

- **Desarrollo de aplicaciones informáticas:** cuáles son los propósitos para aprender a desarrollar software y en qué ámbitos podemos aplicarlos.
- **Introducción a la programación:** metodologías y ciclo de vida de un desarrollo. Análisis, diseño e implementación del sistema. Pseudocódigo y estructuras de control.
- **Primer proyecto en Visual Basic y C++:** características más importantes que encierra cada lenguaje y cómo funciona su interacción con los usuarios.
- **Estructuras de datos:** un vistazo a las estructuras de datos más utilizadas en la programación de cualquier lenguaje y su desarrollo lógico.
- **Normas en la confección de interfaces:** pautas útiles a tener en cuenta para lograr una interfaz funcional y armónica.

SOBRE EL AUTOR

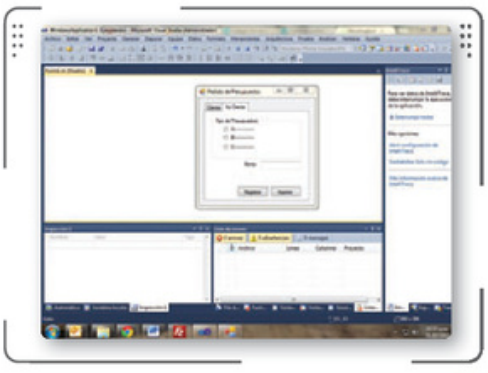
Juan Carlos Casale es Analista de Sistemas y Administrador de Empresas, y da clases de Informática en el Colegio Universitario IES Siglo 21, ubicado en Córdoba capital, Argentina. Allí también edita textos interactivos de estudio y trabaja como coordinador de área en los laboratorios de Informática.

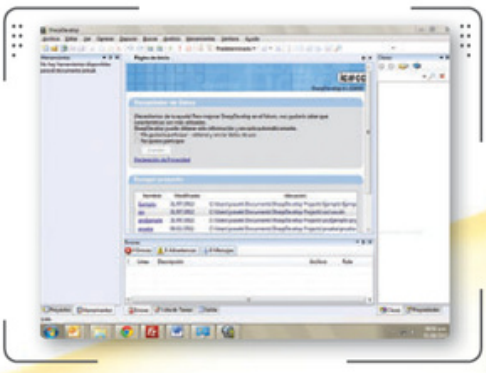
NIVEL DE USUARIO

Básico / Intermedio

CATEGORÍA

Desarrollo





ISBN 978-987-1857-69-2

RU RedUSERS

REDUSERS.com
En nuestro sitio podrá encontrar noticias relacionadas y también participar de la comunidad de tecnología más importante de América Latina.

PROFESOR EN LÍNEA
Ante cualquier consulta técnica relacionada con el libro, puede contactarse con nuestros expertos: profesor@redusers.com.