

Programación en

Zig

Fundamentos Básicos



```
const std = @import("std");  
  
pub fn main() !void {  
    const stdout = std.io.getStdOut();  
    try stdout.writeAll("¡Hola,  
        Zig!\n");  
}
```

Luis Eduardo
Muñoz Guerrero

PÁGINA LEGAL

Título: Programación en Zig: Fundamentos Básicos

ISBN: 978-628-97116-4-6

Sello editorial: Corporación Centro Internacional de Marketing Territorial para la Educación y el Desarrollo (978-628-97116)

Materia: Sistemas

Clasificación THEMA: Lenguajes de programación y extensión/scripting: general

Público objetivo: Enseñanza universitaria o superior

Idioma: Español / Castellano

Número de edición: 1

Ciudad de edición: Medellín

Departamento: Antioquia

Tipo de soporte: Libro digital descargable

Formato: PDF (.pdf)

Protección técnica: ninguna

Tipos de acceso: digital: descarga y en línea

Tipo de contenido: Texto (legible a simple vista)

Número de identificación tributaria o cédula de ciudadanía: 8110433950

Representante legal: Roger Loaiza Álvarez

Responsable ISBN: Juliana Escobar Gómez

e-mail: editorialcimted@gmail.com

Teléfono: 3245664447





Cuidado de la edición:
Juliana Escobar Gómez
Calle 41 no 80b 120 int 301
Código postal: 050031 - Medellín - Colombia
Julio 2026
© Derechos reservados

© Prohibida la reproducción parcial o total sin la previa autorización del sello editorial Centro Internacional de Marketing Territorial para la Educación y el Desarrollo, julio 2026

Las opiniones expresadas y el contenido de este libro son exclusivamente responsabilidad del autor y no indican necesariamente el punto de vista de la Corporación CIMTED

Todo el contenido de este libro está protegido por la ley según los derechos materiales e intelectuales del editor y del autor que escribió este libro, por lo tanto, no está permitido copiar, fragmentar con propósitos comerciales todo su contenido, sin la respectiva autorización de los anteriores.

Si se hace como servicio académico o investigativo, debe contar igualmente con el permiso escrito del autor y citar sus respectivas fuentes

Más información: editorialcimted@gmail.com

Publicación electrónica, editada en Colombia



TABLA DE CONTENIDO

Página legal	2
Tabla de contenido	4
Tabla de tablas	6
Tabla de figuras	9
Capítulo 0	10
Introducción general y propósito del libro	10
Capítulo 1	11
Introducción a la programación y Zig	11
Ejercicios	14
Capítulo 2	16
Instalación y configuración del Entorno	16
Ejercicios	29
Capítulo 3	32
Variables y Tipos de Datos	32
Ejercicios	45
Capítulo 4	48
Operadores y expresiones	48
Ejercicios	64
Capítulo 5	66
Estructuras de control	66
Ejercicios	84
Capítulo 6	86
Funciones	86
Ejercicios	100
Capítulo 7	102
Arrays y Slices	102
Programación en Zig	4

Ejercicios	117
Capítulo 8	118
Punteros y Memoria	118
Ejercicios	135
Capítulo 9	136
Estructuras y Tipos Personalizados	136
Ejercicios	153
Capítulo 10	155
Manejo de Errores	155
Ejercicios	171
Capítulo 11	173
Comptime y Metaprogramación	173
Ejercicios	191
Capítulo 12	192
Entrada/Salida y Archivos	192
Ejercicios	210
Capítulo 13	212
Proyecto final integrador	212
Ejercicios	228
Glosario	229
Bibliografía	231

TABLA DE TABLAS

Tabla 1 - Principios Zen de Zig (Fuente: Propia)	11
Tabla 2 - Pasos de Configuración del Entorno (Fuente: Propia)	16
Tabla 3 Resumen de Instalación por Sistema Operativo (Fuente: Propia)	21
Tabla 4 - Características de un Buen Editor de Código	23
Tabla 5 - Comandos Básicos de Zig (Fuente: Propia)	27
Tabla 6 - Diferencia entre const y var (Fuente: Propia)	33
Tabla 7 - Tipos de Enteros en Zig (Fuente: Propia)	36
Tabla 8 - Tipos de Punto Flotante (Fuente: Propia)	37
Tabla 9 - Caracteres de Escape Comunes (Fuente: Propia)	40
Tabla 10 - Operadores Aritméticos (Fuente: Propia)	49
Tabla 11 - Operadores de Comparación (Fuente: Propia)	53
Tabla 12 - Operadores Lógicos (Fuente: Propia)	55
Tabla 13 - Operadores de Asignación Compuestos (Fuente: Propia)	56
Tabla 14 - Operadores de Bits (Fuente: Propia)	57
Tabla 15 - Estructuras de Control en Zig (Fuente: Propia)	66
Tabla 16 - Variantes de la Sentencia if (Fuente: Propia)	69
Tabla 17 - Control de Bucles: break vs continue (Fuente: Propia)	76
Tabla 18 - Comparación: while vs for (Fuente: Propia)	79
Tabla 19 - Componentes de una Función (Fuente: Propia)	87
Tabla 20 - Tipos de Retorno de Funciones (Fuente: Propia)	88
Tabla 21 - Visibilidad de Funciones (Fuente: Propia)	93
Tabla 22 - Visibilidad de Funciones (Fuente: Propia)	93
Tabla 23 - Funciones Integradas Comunes (Fuente: Propia)	94
Tabla 24 - Arrays vs Slices (Fuente: Propia)	102
Tabla 25 - Formas de Crear Arrays (Fuente: Propia)	104
Tabla 26 - Sintaxis de Slicing (Fuente: Propia)	107

Tabla 27 - Funciones Útiles de std.mem (Fuente: Propia)	111
Tabla 28 - Algoritmos Comunes en Arrays (Fuente: Propia)	114
Tabla 29 - Patrones de Uso de Arrays	115
Tabla 30 - Tipos de Punteros en Zig (Fuente: Propia)	120
Tabla 31 - Operadores de Punteros (Fuente: Propia)	120
Tabla 32 - Punteros Regulares vs Opcionales (Fuente: Propia)	126
Tabla 33 - Stack vs Heap (Fuente: Propia)	127
Tabla 34 - Alineación por Tipo (Fuente: Propia)	131
Tabla 35 - Herramientas de Debugging de Memoria (Fuente: Propia)	132
Tabla 36 - Manejo de Memoria: Zig vs Otros (Fuente: Propia)	133
Tabla 37 - Tipos Personalizados en Zip (Fuente: Propia)	136
Tabla 38 - Parámetro self en Métodos (Fuente: Propia)	141
Tabla 39 – Comparación de Tipos Personalizados (Fuente: Propia)	146 146
Tabla 40 - Patrones de Diseño con Structs (Fuente: Propia)	151
Tabla 41 - Tipos de Struct para Interop (Fuente: Propia)	152
Tabla 42 - Palabras Clave para Errores (Fuente: Propia)	155
Tabla 43 - Sintaxis de Error Unions (Fuente: Propia)	157
Tabla 44 - Patrones Comunes de Manejo de Errores (Fuente: Propia)	164
Tabla 45 - Errores Comunes y Cómo Evitarlos (Fuente: Propia)	168
Tabla 46 - Funciones de Testing para Errores	169
Tabla 47 - Usos de comptime (Fuente: Propia)	173
Tabla 48 - ¿Qué se evalúa en compile-time? (Fuente: Propia)	178
Tabla 49 - Funciones de Reflexion (Fuente: Propia)	180
Tabla 50 - Comptime vs Otras tecnologías (Fuente: Propia)	186
Tabla 51 - Limitaciones de comptime (Fuente: Propia)	187
Tabla 52 - Streams Estándar (Fuente: Propia)	192
Tabla 53 - Operaciones del Sistema de Archivos (Fuente: Propia)	201

Tabla 54 - Conceptos Avanzados de I/O (Fuente: Propia)	206
Tabla 55 - Formatos de Datos Comunes (Fuente: Propia)	207
Tabla 56 - Comandos del Gestor de Tareas (Fuente: Propia)	212

TABLA DE FIGURAS

Figura 1 - Proceso de compilación	17
Figura 2 - Visualización del Array hola_bytes	40
Figura 4 - Flujo de decisión if/else	67
Figura 5 - Flujo de bucle while	74
Figura 6 - Máquina de funciones	87
Figura 7 - Visualización del Array números	103
Figura 8 - Slice como parámetro	108
Figura 9 - Diagrama de puntero ptr	119
Figura 10 - Puntero en función	124
Figura 11 - Puntero opcional	126
Figura 12 - Diagrama de struct Libro	137
Figura 13 - Unión etiquetada	145
Figura 14 - Struct vs enum vs unión	146
Figura 15 - Flujo de errores	156
Figura 16 - Error unions	157
Figura 17 - Errdefer	161
Figura 18 - Comptime overview	174
Figura 19 - Especialización genérica	176
Figura 20 - Type factory	182
Figura 21 - Streams estándar	194
Figura 23 - Estrategias de lectura	198
Figura 24 - Diagrama de struct Tarea	216
Figura 25 - Flujo de comandos	225

CAPÍTULO 0

Introducción general y propósito del libro

Este libro introduce los fundamentos de la programación utilizando Zig como lenguaje de aprendizaje. A través de una formación progresiva, el lector comprenderá no solo la sintaxis del lenguaje, sino también los principios conceptuales que sustentan la escritura de programas claros, correctos y estructurados. El contenido abarca desde variables y tipos de datos hasta el desarrollo de un proyecto final integrador, pasando por operadores, estructuras de control, funciones, manejo de memoria, gestión de errores y operaciones de entrada y salida.

La obra está dirigida principalmente a estudiantes, autodidactas y personas interesadas en iniciar su aprendizaje en programación mediante un lenguaje moderno, explícito y estructurado. También resulta útil para lectores con conocimientos básicos que deseen acercarse a Zig desde una perspectiva introductoria. Aunque ofrece una aproximación amplia a los fundamentos del lenguaje, no profundiza en temas avanzados como optimización compleja, concurrencia, desarrollo de compiladores o arquitecturas de software de escala industrial.

Este libro ha sido elaborado con Zig 0.13.0; por ello, se recomienda verificar que el entorno de trabajo corresponda a esta versión, ya que algunos aspectos del lenguaje pueden variar en futuras actualizaciones. La elección de Zig como lenguaje de enseñanza no es casual: se trata de un lenguaje compilado y moderno que favorece el control directo sobre el comportamiento del programa, evita mecanismos ocultos y promueve una sintaxis transparente. Estas características lo convierten en una opción pertinente para aprender fundamentos de programación de manera clara y consciente.

Para aprovechar este libro, se recomienda tener nociones básicas sobre el uso del computador, la gestión de archivos y el manejo de la terminal. No se requiere experiencia previa con Zig ni conocimientos anteriores en programación, aunque una noción elemental de lógica y resolución de problemas facilitará el aprendizaje. Al completar este libro, el lector habrá desarrollado una comprensión progresiva de los fundamentos de Zig y habrá fortalecido habilidades relacionadas con el pensamiento lógico, la organización de problemas y el análisis de errores.

CAPÍTULO I

Introducción a la programación y Zig

¡Bienvenido al mundo de la programación con Zig! Si estás leyendo este libro, probablemente tengas curiosidad por aprender a crear software, resolver problemas mediante código o explorar un lenguaje moderno y eficiente. Este primer capítulo sentará las bases para comprender qué significa programar y por qué Zig ha sido elegido como lenguaje de aprendizaje en este recorrido.

En esencia, programar consiste en dar instrucciones a una computadora para que realice una tarea específica. Las computadoras son increíblemente rápidas y precisas, pero también completamente literales: no pueden adivinar tus intenciones. Por eso, es necesario indicarles exactamente qué deben hacer, paso a paso.

Un algoritmo es una secuencia ordenada de pasos lógicos orientados a resolver un problema. Puede compararse con una receta de cocina: si se siguen los pasos en el orden adecuado, se obtiene el resultado esperado. Antes de escribir una sola línea de código, es importante tener claro el algoritmo que guiará la solución.

Zig es un lenguaje de programación de sistemas moderno, diseñado para ser robusto, eficiente y claro. A diferencia de otros lenguajes que ocultan parte de lo que ocurre durante la ejecución del programa, Zig valora la transparencia y permite comprender con mayor precisión cómo se comporta el código.

Principio	Descripción
Sin flujo de control oculto	No hay magia: hace exactamente lo que lees. Si no ves una llamada a función, no hay código ejecutándose a escondidas.
Sin asignaciones ocultas	El manejo de memoria es explícito.
Portabilidad	Funciona en casi cualquier sistema operativo y arquitectura.
Mantenibilidad	El código es fácil de leer y entender para otros (y para ti mismo en el futuro).
Simplicidad	El lenguaje es pequeño y fácil de aprender por completo.

Tabla 1 - Principios Zen de Zig
(Fuente: *Propia*)

Tradicionalmente, el primer programa que se escribe al aprender un lenguaje de programación es el conocido **“Hola, mundo”**. Se trata de un programa sencillo que imprime un saludo en la pantalla. A continuación, veamos cómo quedaría el archivo `hello_world.zig`:

1	<code>const std = @import("std");</code>
2	
3	<code>pub fn main() void {</code>
4	<code> std.debug.print("¡Hola, mundo!\n", .{});</code>
5	<code>}</code>

Video 1: Introducción a la programación y zip



Analicemos este código línea por línea:

- `const std = @import("std");` -> Importa la librería estándar de Zig, que contiene funciones útiles.
- `pub fn main() void {` -> Define la función principal donde comienza la ejecución del programa.

- `std.debug.print(...)` -> Llama a una función para imprimir texto en la consola.
- `.{}` -> Es una lista de argumentos vacía. Zig requiere esto incluso si no hay variables en el mensaje.
- `}` -> Cierra el bloque de la función.

Zig es un lenguaje compilado. Esto significa que el código fuente, escrito de forma legible para los seres humanos, debe traducirse a código máquina, que es el formato que la computadora puede ejecutar. Esta traducción se realiza mediante un programa llamado **compilador**.

El comando básico para ejecutar un programa en Zig es:

1

```
zig run hello_world.zig
```

EJERCICIOS

Pon a prueba lo aprendido con los siguientes ejercicios:

TU PRIMER PROGRAMA PERSONALIZADO

Escribe un programa similar a "Hola, mundo", pero haz que imprima tres líneas diferentes: la primera con tu nombre, la segunda con tu ciudad o país, y la tercera con tu pasatiempo favorito. Usa el carácter especial `\n` al final de cada texto para asegurarte de que las líneas se impriman por separado.

Ejercicio

LAS 6 ETAPAS APLICADAS

Piensa en un programa hipotético que calcule el promedio de tus calificaciones. Describe brevemente qué harías en cada una de las seis etapas del desarrollo de software: análisis, diseño, implementación, pruebas, despliegue y mantenimiento.

Como apoyo, investiga especialmente las etapas de despliegue y mantenimiento, ya que suelen ser menos conocidas al iniciar en programación.

Ejercicio

COMENTARIOS EN EL CÓDIGO

Escribe un programa que tenga comentarios explicativos. Los comentarios en Zig comienzan con '//'. Agrega comentarios antes de la importación, antes de la función main, y dentro de la función explicando qué hace el print.

Ejercicio

3

ERRORES DE SINTAXIS INTENCIONALES

Copia el código del ejercicio 1 y elimina intencionalmente el punto y coma ';' al final de una de las líneas. Intenta compilarlo con 'zig run'. ¿Qué mensaje de error te da el compilador? Corrige el error y prueba de nuevo.

Ejercicio

4

CAPÍTULO 2

INSTALACIÓN Y CONFIGURACIÓN DEL ENTORNO

Antes de escribir y ejecutar programas en Zig, es necesario preparar el entorno de trabajo. Aunque este proceso puede parecer tedioso al principio, constituye un paso fundamental que todo programador debe aprender a realizar. La buena noticia es que solo tendrás que hacerlo una vez; después, tu computadora quedará lista para programar en Zig cuando lo necesites.

Este capítulo te guiará paso a paso durante el proceso de instalación y configuración. Además de indicar qué debes hacer, explicará por qué cada paso es importante.

El proceso de configuración incluye tres partes principales. Primero, descargaremos e instalaremos el compilador de Zig, que es el programa encargado de traducir el código escrito por el programador a instrucciones que la computadora puede ejecutar.

Segundo, elegiremos e instalaremos un editor de código, es decir, un programa especializado que facilita la escritura de código de forma cómoda y productiva. Tercero, verificaremos que todo funcione correctamente mediante la escritura y ejecución de un primer programa real. Al finalizar este capítulo, habrás ejecutado código Zig en tu propia computadora.

Paso	Acción	Resultado
1	Instalar el compilador de Zig	Traducción del código a un ejecutable
2	Configurar un editor de código	Escritura cómoda y organizada del código
3	Verificar la instalación	Ejecución del primer programa

Tabla 2 - Pasos de Configuración del Entorno
(Fuente: Propia)

Antes de comenzar con la instalación, conviene entender qué es un compilador y por qué es necesario. Como se mencionó en el capítulo anterior, las computadoras solo entienden código máquina, es decir, secuencias de unos y ceros que serían muy difíciles de escribir directamente. El compilador actúa como un traductor: toma el código escrito en Zig, relativamente fácil de leer para los seres humanos, y lo convierte en código máquina que la computadora puede ejecutar.

Al principio, esta exigencia puede parecer molesta; sin embargo, en realidad es una característica muy valiosa. Es preferible descubrir un error al compilar el programa que encontrarlo cuando ya se está ejecutando y algo falla de manera inesperada.

El compilador de Zig es conocido por la calidad de sus mensajes de error. A diferencia de otros compiladores que generan mensajes crípticos o confusos, el compilador de Zig procura indicar con claridad qué está mal y, en muchos casos, sugiere cómo corregirlo.

Por ejemplo, si escribes incorrectamente el nombre de una función, el compilador no solo indicará que esa función no existe, sino que también podrá sugerir funciones con nombres similares a la que probablemente querías usar.

Esta característica puede ahorrarte muchas horas de frustración durante el proceso de aprendizaje.

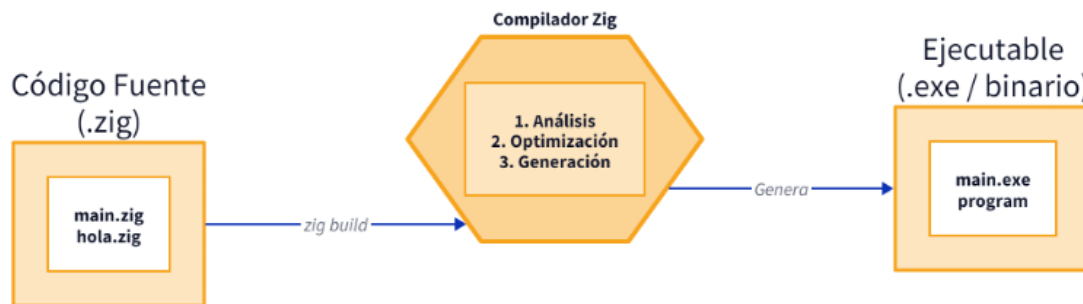


Figura 1 - Proceso de compilación

Comencemos con la instalación en Windows, que es el sistema operativo más común. El proceso es sencillo, pero requiere algunos pasos que explicaré en detalle. Lo primero que debes hacer es abrir tu navegador web y visitar la página oficial de Zig: ziglang.org. Esta es la fuente oficial y siempre debes descargar Zig desde aquí para asegurarte de obtener una versión legítima y segura.

Una vez en la página principal, busca el enlace que dice "Download" o "Descargas" en el menú de navegación. Al hacer clic, llegarás a una página que muestra todas las versiones disponibles de Zig. Verás varias secciones para diferentes sistemas operativos. En la sección de Windows, encontrarás varios archivos disponibles. El que necesitas es el que termina en "-windows-x86_64.zip". Este es el compilador para sistemas Windows de 64 bits, que es lo que tiene la gran mayoría de computadoras modernas. El archivo tiene un tamaño de aproximadamente 50-70 megabytes.

Después de descargar el archivo, ve a tu carpeta de Descargas y busca el archivo zip. Haz clic derecho sobre él y selecciona "Extraer todo" o "Extract All". Windows te preguntará dónde quieres extraer los archivos. Aquí viene una decisión importante: te recomiendo extraerlo

directamente en la raíz de tu disco C:, creando una carpeta llamada simplemente "zig". Esto significa que la ruta final sería C:\zig. La razón de usar una ruta corta y simple es que facilitará los siguientes pasos y evitará posibles problemas con rutas muy largas o con espacios.

Después de extraer el archivo, abre la carpeta C:\zig (o donde hayas decidido extraerlo). Dentro encontrarás varios archivos y carpetas. El más importante es "zig.exe", que es el compilador propiamente dicho. También verás carpetas como "lib" que contiene la librería estándar, y "doc" que contiene documentación. No necesitas hacer nada con estos archivos por ahora, solo asegúrate de que están ahí.

Ahora viene el paso más importante y a veces más confuso para los principiantes: agregar Zig al PATH del sistema. El PATH es una lista de carpetas donde Windows busca programas cuando escribes un comando en la terminal.

Sin este paso, cada vez que quieras usar Zig tendrías que escribir la ruta completa C:\zig\zig.exe, lo cual sería muy incómodo. Al agregar Zig al PATH, podrás simplemente escribir "zig" desde cualquier carpeta y Windows sabrá dónde encontrarlo.

Para modificar el PATH en Windows 10 y 11, sigue estos pasos: Primero, presiona la tecla Windows en tu teclado y escribe "variables de entorno". Verás aparecer una opción que dice "Editar las variables de entorno del sistema" o algo similar. Haz clic en esa opción. Se abrirá una ventana llamada "Propiedades del sistema". En la parte inferior de esta ventana, haz clic en el botón "Variables de entorno".

En la nueva ventana que aparece, verás dos secciones: "Variables de usuario" en la parte superior y "Variables del sistema" en la parte inferior. Busca la variable llamada "Path" en la sección de "Variables del sistema" (la inferior). Selecciónala haciendo clic una vez y luego haz clic en "Editar". En la ventana que aparece, verás una lista de rutas. Haz clic en "Nuevo" y escribe la ruta donde instalaste Zig. Si seguiste mi recomendación, escribirías C:\zig. Haz clic en "Aceptar" en todas las ventanas abiertas para guardar los cambios.



NOTA

Los cambios en el PATH no afectan a las ventanas de terminal que ya tenías abiertas. Necesitas abrir una nueva ventana de PowerShell o Símbolo del sistema para que los cambios surtan efecto. Si Zig sigue sin funcionar después de modificar el PATH, intenta reiniciar tu computadora.

Existe una alternativa más fácil si te sientes cómodo usando gestores de paquetes. Scoop es un gestor de paquetes para Windows que simplifica enormemente la instalación de herramientas de desarrollo. Si tienes Scoop instalado, puedes instalar Zig simplemente abriendo PowerShell y ejecutando el comando "scoop install zig". Scoop se encargará de descargar, instalar y configurar el PATH automáticamente. Si no tienes Scoop y quieres instalarlo, puedes encontrar instrucciones en scoop.sh.

Si estás usando Linux, el proceso de instalación es ligeramente diferente pero igual de sencillo. Linux usa la terminal mucho más que Windows, así que la mayor parte de la instalación se hará escribiendo comandos.

Primero, abre tu navegador y ve a ziglang.org/download. Busca la sección para Linux. Encontrarás un archivo que termina en "-linux-x86_64.tar.xz". Este es un archivo comprimido similar al zip de Windows, pero usando un formato de compresión diferente que es más común en Linux. Descarga este archivo, que probablemente irá a tu carpeta de Descargas.

Ahora abre una terminal. Puedes hacerlo buscando "Terminal" en el menú de aplicaciones, o presionando Ctrl+Alt+T en la mayoría de las distribuciones. Una vez en la terminal, necesitamos navegar a la carpeta de Descargas. Escribe el siguiente comando y presiona Enter:

1

```
cd ~/Descargas
```

El comando `cd` significa *change directory* —cambiar de directorio— y el símbolo `~` es una abreviatura que representa la carpeta personal del usuario. Si tu sistema está en inglés, la carpeta podría llamarse `Downloads` en lugar de `Descargas`; en ese caso, el comando sería:

Ahora extraeremos el archivo descargado. En Linux, el comando `tar` se usa para trabajar con archivos comprimidos. Escribe el siguiente comando, reemplazando el nombre del archivo si es diferente:

```
1 tar -xf zig-linux-x86_64-*.tar.xz
```

El asterisco (*) es un comodín que coincide con cualquier versión, así que este comando funcionará independientemente de qué versión específica hayas descargado. Después de ejecutar este comando, tendrás una nueva carpeta con un nombre como "zig-linux-x86_64-0.13.0" (el número de versión puede variar).

Ahora moveremos esta carpeta a una ubicación más permanente. En Linux, las aplicaciones instaladas manualmente suelen ir en la carpeta `/opt`. Usaremos el comando `mv` (move) con `sudo` (que te da permisos de administrador):

```
1 sudo mv zig-linux-x86_64-* /opt/zig
```

Finalmente, debemos agregar Zig al `PATH`. En Linux, esto se hace editando un archivo de configuración ubicado en la carpeta personal del usuario. Si usas Bash, el intérprete de comandos predeterminado en muchas distribuciones, el archivo es `~/.bashrc`. Si usas Zsh, el archivo es `~/.zshrc`. Para este ejemplo, usaremos `nano`, un editor de texto sencillo que suele venir preinstalado en la mayoría de los sistemas:

```
1 nano ~/.bashrc
```

Esto abrirá el editor de texto `nano` dentro de la terminal. Usa las flechas del teclado para ir hasta el final del archivo (o presiona `Ctrl+End` para ir directamente al final). Agrega la siguiente línea al final del archivo:

```
1 export PATH=$PATH:/opt/zig
```

Esta línea le dice al sistema que agregue `/opt/zig` a la lista de carpetas donde buscar programas. Después de agregar la línea, presiona `Ctrl+O` para guardar (luego `Enter` para confirmar) y `Ctrl+X` para salir de `nano`. Finalmente, para que los cambios surtan efecto sin reiniciar, ejecuta:

```
1 source ~/.bashrc
```

Muchas distribuciones de Linux también ofrecen Zig en sus repositorios oficiales de paquetes. En Ubuntu o Debian puedes probar "sudo apt install zig". En Fedora usa "sudo dnf

install zig". En Arch Linux usa "sudo pacman -S zig". La ventaja de usar el gestor de paquetes es que es más fácil y se integra mejor con el sistema. La desventaja es que a veces las versiones en los repositorios están desactualizadas. La instalación manual desde el sitio oficial garantiza que tengas la versión más reciente.

Para los usuarios de macOS, la forma más sencilla de instalar Zig es usando Homebrew, el gestor de paquetes más popular para Mac. Si ya tienes Homebrew instalado, la instalación de Zig es increíblemente simple: abre la Terminal (puedes encontrarla en Aplicaciones > Utilidades > Terminal, o buscándola con Spotlight presionando Cmd+Espacio) y ejecuta:

1

brew install zig

Homebrew se encargará de todo: descargará Zig, lo instalará en la ubicación correcta, y configurará el PATH automáticamente. Después de unos minutos, Zig estará listo para usar. Si no tienes Homebrew, puedes instalarlo visitando brew.sh y siguiendo las instrucciones.

Si prefieres no usar Homebrew o no puedes instalarlo, el proceso manual es muy similar al de Linux. Descarga el archivo para macOS desde ziglang.org/download (será algo como "zig-macos-x86_64-*.tar.xz" o "zig-macos-aarch64-*.tar.xz" si tienes una Mac con chip Apple Silicon M1/M2). Extrae el archivo con el comando tar, mueve la carpeta a /opt/zig o a otra ubicación de tu preferencia, y agrega la ruta al PATH editando ~/.zshrc (que es el archivo de configuración por defecto en macOS moderno).

Sistema	Método Rápido	Archivo de descarga
Windows	scoop install zig	zig-windows-x86_64.zip
Linux	apt/dnf/pacman install zig	zig-linux-x86_64.tar.xz
macOS	brew install zig	zig-macos-x86_64.tar.xz

Tabla 3 Resumen de Instalación por Sistema Operativo
(Fuente: Propia)

Independientemente de qué sistema operativo uses y que método de instalación hayas elegido, es crucial verificar que la instalación fue exitosa antes de continuar. Abre una nueva ventana de terminal (esto es importante: debe ser una ventana nueva, no una que ya tenías abierta) y ejecuta el siguiente comando:

1

zig version

Si todo está bien configurado, verás un mensaje mostrando la versión de Zig que tienes instalada, algo como "0.13.0" o similar. El número exacto dependerá de cuándo hagas la instalación. Si recibes un error diciendo que el comando no se reconoce o no se encuentra, significa que algo salió mal con la configuración del PATH. Revisa los pasos anteriores cuidadosamente, asegurándote de que escribiste las rutas correctamente y de que estás usando una ventana de terminal nueva.



Consejo

Guarda en un lugar seguro el resultado del comando `zig version`. Cuando busques ayuda en foros o comunidades en línea, saber exactamente qué versión estás usando te permitirá obtener respuestas más precisas.

Ten en cuenta que diferentes versiones de Zig pueden presentar pequeñas variaciones en la sintaxis o en las funciones disponibles.

Ahora que Zig está instalado, necesitamos un buen editor de código. En teoría, podrías escribir código Zig en cualquier editor de texto, incluso en el Bloc de notas de Windows o en TextEdit de macOS. Sin embargo, hacerlo sería como intentar escribir una novela en una máquina de escribir antigua cuando tienes disponible una computadora moderna con procesador de texto.

Los editores de código especializados ofrecen características que hacen que la programación sea mucho más agradable y productiva.

La característica más evidente de un editor de código es el resaltado de sintaxis. Esto significa que el editor usa diferentes colores para distinguir las partes del código: las palabras clave del lenguaje, los textos entre comillas, los comentarios, entre otros elementos. Este

coloreado hace que el código sea más fácil de leer y ayuda a detectar ciertos errores visualmente. Por ejemplo, si olvidas cerrar unas comillas, todo lo que sigue aparecerá con un color incorrecto, lo que puede ayudarte a identificar el problema.

Otra característica importante es el autocompletado. Mientras escribes, el editor puede sugerir palabras, funciones o estructuras que podrías necesitar. Esto no solo acelera la escritura, sino que también ayuda a descubrir funciones nuevas y reduce los errores en los nombres de funciones y variables. Algunos editores incluso muestran documentación sobre las funciones mientras escribes, indicando qué hacen y qué parámetros aceptan.

Los editores de código también ofrecen detección de errores en tiempo real. Antes de intentar compilar el programa, el editor puede subrayar partes del código que presentan problemas, de manera similar a como un procesador de texto subraya las palabras mal escritas. Esto permite corregir errores sobre la marcha y evita repetir innecesariamente el ciclo de compilar, revisar el error, buscar su ubicación y corregirlo.

Características	Descripción	Beneficio
Resaltado de sintaxis	Usa colores para distinguir los elementos del código	Código más legible
Autocompletado	Ofrece sugerencias mientras escribes	Menos errores y mayor velocidad
Detección de errores	Subraya problemas en tiempo real	Corrección inmediata
Formateo automático	Alinea y organiza el código al guardar	Código más ordenado

Tabla 4 - Características de un Buen Editor de Código
(Fuente: Propia)

Mi recomendación para principiantes es Visual Studio Code, también conocido como VS Code. Es gratuito, funciona en Windows, Linux y macOS, tiene buen soporte para Zig y es uno de los editores de código más populares en la actualidad.

Para instalarlo, visita `code.visualstudio.com` y descarga la versión correspondiente a tu sistema operativo. La instalación es sencilla: ejecuta el instalador descargado y sigue las instrucciones en pantalla, aceptando las opciones predeterminadas.

Una vez instalado VS Code, necesitas agregar la extensión de Zig para obtener las funcionalidades específicas del lenguaje. Las extensiones son pequeños complementos que añaden características adicionales a VS Code.

Para instalar la extensión de Zig, abre VS Code y observa la barra lateral izquierda. Allí verás varios íconos; haz clic en el que tiene forma de cuatro cuadrados pequeños, correspondiente a la sección de **Extensiones**. También puedes abrirla con el atajo `Ctrl + Shift + X`.

En el buscador de extensiones, escribe `zig` y busca una extensión relacionada con el lenguaje Zig, como **Zig Language** o **Zig Language Server**. Haz clic en **Install** para instalarla.

La extensión de Zig proporciona resaltado de sintaxis específico para el lenguaje, formateo automático del código, integración con el compilador para mostrar errores directamente en el editor y autocompletado básico. Estas características harán que la experiencia de programación sea más fluida. Para verificar que funciona correctamente, escribe `std.` en tu archivo y espera un momento; debería aparecer una lista desplegable con opciones.

Otros editores populares que también funcionan bien con Zig son Sublime Text, Vim, Neovim y Emacs. Si ya tienes experiencia con alguno de ellos y te sientes cómodo usándolo, busca e instala el complemento o la extensión correspondiente para Zig.

Sin embargo, si estás comenzando, se recomienda usar VS Code por su interfaz amigable, su facilidad de configuración y su amplia comunidad de usuarios.



NOTA

Con Zig instalado y VS Code configurado, es momento de escribir y ejecutar tu primer programa real. No se trata del programa que vimos en el capítulo anterior solo como ejemplo de lectura, sino de uno que escribirás y ejecutarás en tu propia computadora.

Primero, crea una carpeta para organizar tus proyectos de Zig. Puedes crearla donde prefieras, pero se recomienda elegir una ubicación fácil de encontrar. En Windows, podrías crearla en `C:\proyectos\zig` o en la carpeta de Documentos. En Linux o macOS, una buena opción sería `~/proyectos/zig`.

Puedes crear esta carpeta desde el explorador de archivos o desde la terminal con los siguientes comandos:

```
mkdir proyectos
mkdir proyectos/zig
```

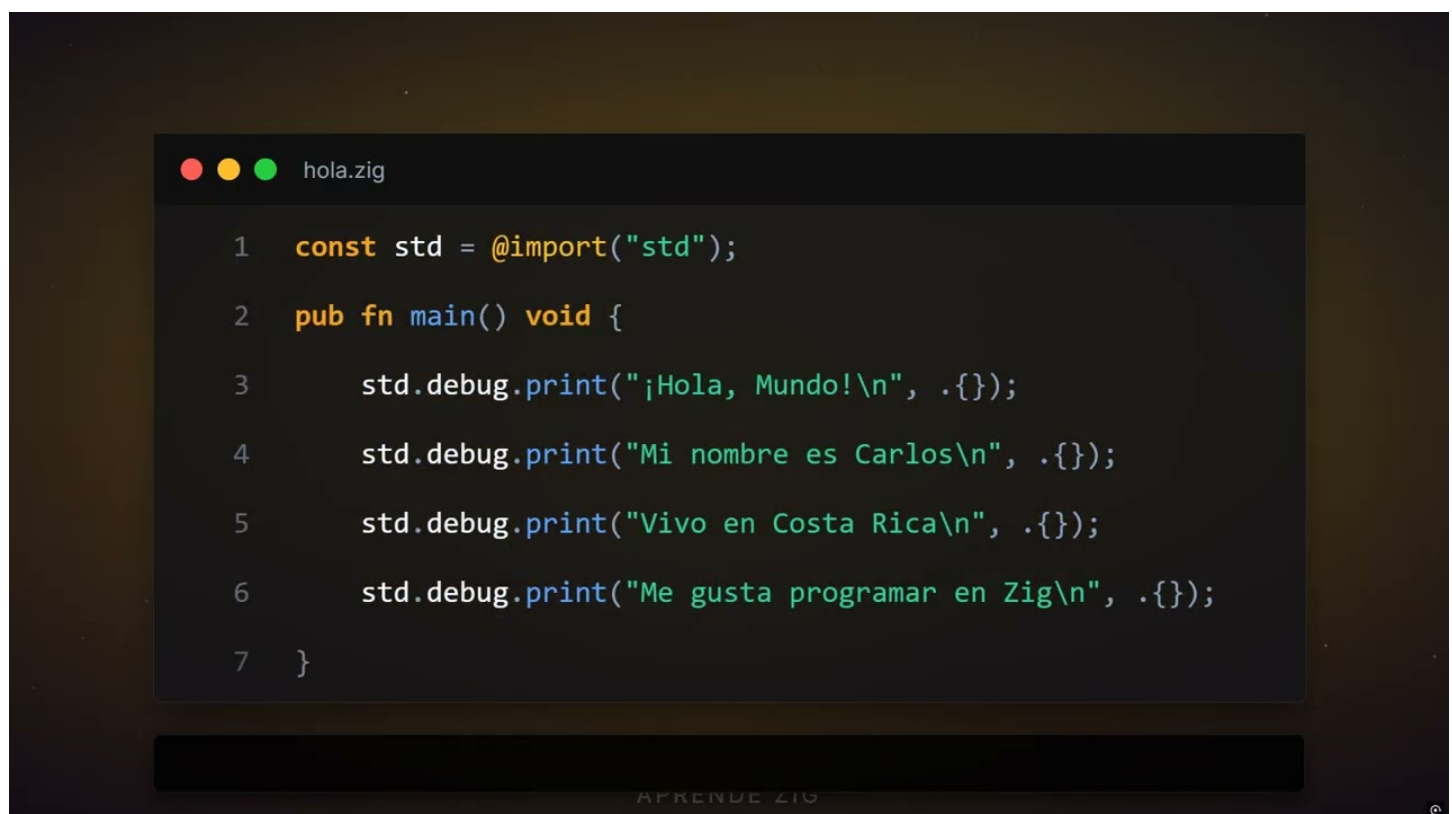
Ahora abre VS Code. Ve al menú **Archivo** o **File** y selecciona **Abrir carpeta** u **Open Folder**. Navega hasta la carpeta que acabas de crear y ábrela. VS Code mostrará la carpeta en el panel izquierdo, que por ahora estará vacío.

Para crear tu primer archivo, haz clic derecho en el panel izquierdo, donde aparece el nombre de tu carpeta, y selecciona **Nuevo archivo** o **New File**. Nómbralo `hola.zig`. Es importante incluir la extensión `.zig` para que VS Code y el compilador reconozcan que se trata de un archivo Zig.

En el archivo recién creado, escribe el siguiente código. Tómate tu tiempo y escríbelo cuidadosamente, prestando atención a cada detalle: las mayúsculas y minúsculas importan, los puntos y comas son necesarios, y los paréntesis deben coincidir correctamente.

1	<code>const std = @import("std");</code>
2	
3	<code>pub fn main() void {</code>
4	<code>std.debug.print("¡Hola, Mundo!\n", .{});</code>
5	<code>}</code>

Video 2 - Creación y ejecución del primer programa en Zig

A screenshot of a code editor window titled 'hola.zig'. The code is written in Zig and includes line numbers 1 through 7. The code imports the standard library, defines a main function, and prints four debug messages. The editor has a dark theme with syntax highlighting. At the bottom of the editor, there is a footer that says 'APRENDE ZIG' and a small logo on the right.

```
1  const std = @import("std");
2
3  pub fn main() void {
4      std.debug.print("¡Hola, Mundo!\n", .{});
5      std.debug.print("Mi nombre es Carlos\n", .{});
6      std.debug.print("Vivo en Costa Rica\n", .{});
7      std.debug.print("Me gusta programar en Zig\n", .{});
8  }
```

Después de escribir el código, guarda el archivo presionando `Ctrl+S`, o `Cmd+S` si usas macOS. Si la extensión de Zig está instalada y funcionando correctamente, VS Code mostrará el código con colores diferentes según sus elementos. Si existe algún error de sintaxis, aparecerá una línea roja ondulada debajo del fragmento problemático.

Ahora viene la parte importante: ejecutar el programa. VS Code tiene una terminal integrada que puedes abrir desde el menú Ver (View) > Terminal, o presionando `Ctrl+`` (la tecla de acento grave, generalmente arriba de Tab). La terminal se abrirá en la parte inferior de la ventana, y automáticamente estará ubicada en la carpeta de tu proyecto.

En la terminal, escribe el siguiente comando y presiona Enter:

1	<code>zig run hola.zig</code>
---	-------------------------------

El comando `zig run` le indica al compilador de Zig que compile el programa y luego lo ejecute inmediatamente. Esta opción resulta muy útil durante el desarrollo, porque combina dos pasos en uno.

Después de presionar Enter, deberías ver el mensaje **¡Hola, mundo!** en la terminal. ¡Felicitaciones! Acabas de ejecutar tu primer programa en Zig. Este es un momento importante: has pasado de leer sobre programación a escribir y ejecutar tu propio código.



Consejo

Si recibes un error, no te desanimes. Lee cuidadosamente el mensaje que muestra Zig, ya que suele indicar qué está mal y en qué línea se encuentra el problema. Los errores más comunes al comenzar son olvidar un punto y coma, escribir incorrectamente alguna palabra o no cerrar bien las comillas, los paréntesis o las llaves. Compara tu código con el ejemplo, carácter por carácter.

El comando `zig run` es adecuado para el desarrollo rápido, pero en algunos casos querrás crear un archivo ejecutable independiente. Un ejecutable es un archivo que puede ejecutarse directamente, sin necesidad de usar el compilador de Zig en cada ocasión. Esto

resulta útil cuando deseas compartir tu programa con otras personas o distribuirlo de una forma más profesional.

Para crear un ejecutable, usa el comando `zig build-exe`, seguido del nombre del archivo:

1

zig build-exe hola.zig

Este comando creará un archivo ejecutable en la misma carpeta. En Windows, el archivo se llamará `hola.exe`. En Linux y macOS, se llamará simplemente `hola`, sin extensión.

Una vez creado, puedes ejecutar este archivo directamente. En Windows, escribe uno de los siguientes comandos en la terminal:

En Windows, escribe `.\hola` o `.\hola.exe` en la terminal. En Linux y macOS, escribe `./hola`. El prefijo `./` es necesario en sistemas Unix para indicar que quieres ejecutar un archivo en el directorio actual.

Comando	Uso	Resultado
zig version	Verificar la instalación	Muestra versión instalada
zig run archivo.zig	Desarrollo rápido	Compila y ejecuta inmediatamente
zig build-exe archivo.zig	Crear un ejecutable	Genera archivo .exe (Windows)
zig zen	Filosofía de Zig	Muestra principios de diseño

Tabla 5 - Comandos Básicos de Zig
(Fuente: Propia)

También notarás que se crearon algunos archivos y carpetas adicionales, como una carpeta llamada "zig-cache". Esta carpeta contiene archivos temporales que el compilador usa para acelerar compilaciones futuras. Puedes ignorarla o incluso borrarla si quieres; se recreará automáticamente cuando compiles de nuevo. Algunos programadores agregan "zig-cache" a su archivo `.gitignore` para evitar incluirla en el control de versiones.

A medida que avances en este libro y escribas más programas, es importante mantener tu código organizado. Te recomiendo crear una carpeta separada para cada capítulo del libro, y dentro de ella guardar los ejercicios correspondientes. Por ejemplo, podrías tener una estructura como: `proyectos/zig/capítulo02/hola.zig`, `proyectos/zig/capítulo03/variables.zig`, etc.

Una buena práctica es usar nombres de archivo descriptivos. En lugar de nombres genéricos como "prueba.zig" o "codigo.zig", usa nombres que indiquen qué hace el programa:

"calculadora_suma.zig", "ejercicio_bucles.zig", "juego_adivinanza.zig". Tu yo del futuro te agradecerá cuando estés buscando un ejercicio específico que hiciste hace semanas.

En este capítulo hemos preparado completamente tu entorno de desarrollo para programar en Zig. Instalaste el compilador de Zig en tu sistema operativo, configuraste el PATH para poder usarlo desde cualquier ubicación, instalaste VS Code con la extensión de Zig, y lo más importante: escribiste, compilaste y ejecutaste tu primer programa real. Ya no eres simplemente alguien que lee sobre programación; eres alguien que programa.

En el próximo capítulo comenzaremos a explorar los fundamentos del lenguaje Zig en serio. Aprenderás sobre variables, que son como pequeñas cajas donde puedes guardar información, y sobre los diferentes tipos de datos que Zig puede manejar: números enteros, números decimales, texto, y más. Estos conceptos son los bloques básicos de construcción de cualquier programa, y una vez que los domines, podrás empezar a crear programas que hacen cosas verdaderamente interesantes.

EJERCICIOS

Pon a prueba lo aprendido con los siguientes ejercicios:

VERIFICACIÓN COMPLETA DEL ENTORNO

Abre una terminal y ejecuta los siguientes comandos: 'zig version', 'zig --help', y 'zig zen'. El tercer comando mostrará la filosofía de diseño de Zig en forma de aforismos. Anota la versión de Zig que tienes instalada y escribe tu aforismo favorito de 'zig zen' junto con una breve explicación de lo que crees que significa.

Ejercicio

MODIFICANDO EL MENSAJE

Modifica el programa hola.zig para que muestre un mensaje diferente. Prueba varios mensajes: tu nombre, un saludo en otro idioma, una frase motivacional. Cada vez que hagas un cambio, guarda el archivo y ejecuta 'zig run hola.zig' para ver el resultado.

Ejercicio

MÚLTIPLES MENSAJES

Modifica el programa para que muestre tres líneas diferentes de texto. Cada línea debe mostrar información diferente sobre ti: tu nombre en la primera línea, tu país o ciudad en la segunda, y tu hobby favorito en la tercera

Ejercicio

3

CREANDO UN EJECUTABLE

Usa el comando 'zig build-exe hola.zig' para crear un archivo ejecutable. Encuentra el archivo generado en tu carpeta (hola.exe en Windows, hola en Linux/Mac). Ejecútalo directamente sin usar el comando zig. Luego, copia este ejecutable a otra carpeta o incluso a otra computadora (si tienes acceso a una) y verifica que funciona.

Ejercicio

4

EXPERIMENTANDO CON ERRORES

Este ejercicio te ayudará a entender los mensajes de error de Zig. Haz las siguientes modificaciones intencionalmente incorrectas a tu programa (una a la vez, regresando a la versión correcta después de cada una): (a) Borra el punto y coma al final de la línea del print. (b) Escribe 'Main' en lugar de 'main'. (c) Olvida cerrar las comillas del mensaje. (d) Escribe 'standart' en lugar de 'std'. Para cada error, anota el mensaje de error que Zig te muestra y explica qué te dice.

Ejercicio

5

CONFIGURANDO TU EDITOR PARA ZIG

Si instalaste Visual Studio Code, realiza los siguientes pasos: (a) Abre VS Code y ve a la sección de Extensiones (icono de cuadrados en la barra lateral o Ctrl+Shift+X). (b) Busca 'Zig Language' o 'ZLS' e instala la extensión oficial. (c) Abre tu archivo `hola.zig` y verifica que el código tenga colores (syntax highlighting). (d) Escribe `'std.'` y observa si aparece un menú de autocompletado con opciones. (e) Introduce un error intencionalmente y verifica si el editor lo subraya en rojo antes de compilar. Anota que funcionalidades observas.

Ejercicio

CAPÍTULO 3

Variables y Tipos de Datos

Imagina que estás cocinando una receta compleja que requiere varios ingredientes. Para mantener todo organizado, usas diferentes recipientes: un bol para la harina, otro para el azúcar, una taza medidora para la leche, y así sucesivamente. Cada recipiente tiene un propósito específico, contiene un tipo de ingrediente particular, y tiene un nombre o etiqueta que te ayuda a identificarlo. Las variables en programación funcionan exactamente de la misma manera: son contenedores con nombres que usamos para almacenar información que nuestro programa necesita recordar y utilizar.

Una variable es, en su esencia más simple, un nombre que le damos a un espacio en la memoria de la computadora donde guardamos un valor. Cuando creas una variable, estás reservando un pequeño pedazo de memoria y diciéndole a la computadora: "Guarda este dato aquí, y cuando te pida este nombre, devuélveme el dato que guardaste". Esta capacidad de almacenar y recuperar información es absolutamente fundamental para cualquier programa útil. Sin variables, cada cálculo que hicieras se perdería inmediatamente después de hacerse, como escribir en arena que el viento borra al instante.

Para entender por qué las variables son tan importantes, pensemos en un programa simple que calcula el área de un rectángulo. Necesitas recordar el ancho, recordar el alto, multiplicarlos, y mostrar el resultado. Cada uno de esos valores necesita ser almacenado en algún lugar: una variable para el ancho, otra para el alto, y posiblemente una tercera para el resultado. A medida que los programas se vuelven más complejos, el número de variables crece, y una buena organización de estas variables se vuelve crucial para mantener el código comprensible.

En Zig, las variables se declaran usando las palabras clave "var" o "const". La diferencia entre ellas es fundamental y refleja una filosofía importante del lenguaje: "const" crea una constante, un valor que no puede cambiar después de ser asignado, mientras que "var" crea una variable mutable, cuyo valor puede ser modificado a lo largo del programa. Zig te anima fuertemente a usar "const" siempre que sea posible, porque los valores que no cambian son más fáciles de razonar y menos propensos a errores. De hecho, si declaras una variable con "var" pero nunca la modificas, el compilador te advertirá que deberías usar "const" en su lugar.

Palabra Clave	Tipo	¿Puede cambiar?	Cuándo usar
const	Constante	No	Valores que no cambian (preferido)
var	Variable	Sí	Valores que necesitan modificarse

Tabla 6 - Diferencia entre const y var
(Fuente: Propia)

Veamos cómo se declara una constante en Zig. La sintaxis básica es:

1	<code>const nombre = valor;</code>
---	------------------------------------

Por ejemplo, si queremos guardar nuestra edad en una constante, escribiríamos:

1	<code>const edad = 25;</code>
---	-------------------------------

Con esta línea, hemos creado una constante llamada "edad" que contiene el valor 25. A partir de este momento, cada vez que escribamos "edad" en nuestro código, Zig entenderá que nos referimos al valor 25. El nombre "edad" es lo que llamamos un identificador, y hay ciertas reglas sobre cómo pueden escribirse los identificadores en Zig. Los identificadores pueden contener letras (mayúsculas y minúsculas), dígitos y guiones bajos, pero deben comenzar con una letra o guion bajo, nunca con un número. Así que "edad", "edad_usuario", "Edad2", y "_temporal" son todos identificadores válidos, pero "2edad" o "edad-usuario" (con guion normal) no lo son.

Una característica importante de Zig es que distingue entre mayúsculas y minúsculas. Esto significa que "edad", "Edad", "EDAD" y "EdAd" son todos identificadores diferentes que podrían contener valores distintos. La convención en Zig es usar snake_case para los nombres de variables y funciones, lo que significa escribir todo en minúsculas y separar las palabras con guiones bajos. Este estilo se conoce como snake_case («caso serpiente»), porque las palabras se unen con guiones bajos, evocando los segmentos del cuerpo de una serpiente. Así, en lugar de "edadUsuario" o "EdadUsuario", escribiríamos "edad_usuario". Seguir esta convención hace que tu código sea consistente con el resto del ecosistema Zig y más fácil de leer para otros programadores.

Ahora bien, ¿qué pasa si necesitas un valor que sí puede cambiar? Para eso usamos "var" en lugar de "const":

1	<code>var contador = 0;</code>
2	<code>contador = contador + 1; // Esto es válido</code>
3	<code>contador = 5; // Esto también es válido</code>

La primera línea crea una variable llamada "contador" con un valor inicial de 0. Las siguientes líneas demuestran que podemos cambiar el valor de "contador": primero lo incrementamos en 1 (quedando en 1), y luego lo establecemos directamente a 5. Si intentáramos hacer esto mismo con una constante declarada con "const", el compilador nos daría un error, protegiéndonos de modificar accidentalmente algo que no debíamos modificar.

Cuando el compilador de Zig te dice que deberías usar 'const' en lugar de 'var', no te está regañando; te está ayudando a escribir mejor código. Los valores inmutables (que no cambian) son más seguros porque no hay riesgo de que otra parte del programa los modifique inesperadamente. Sigue el consejo del compilador y usa 'const' por defecto, cambiando a 'var' solo cuando realmente necesites modificar el valor.

NOTA

Hasta ahora hemos hablado de guardar valores en variables, pero no hemos mencionado algo crucial: no todos los valores son iguales. Un número como 42 es muy diferente de un texto como "Hola", que a su vez es diferente de un valor verdadero/falso como "sí" o "no". Estas diferentes categorías de valores se llaman tipos de datos, y son fundamentales en la programación.

Zig es lo que llamamos un lenguaje de tipado estático. Esto significa que cada variable tiene un tipo específico que el compilador conoce antes de ejecutar el programa. El compilador usa esta información para verificar que no estés haciendo cosas sin sentido, como intentar sumar un número y un texto, o dividir un texto entre otro. Estas verificaciones en tiempo de compilación previenen muchos errores que de otro modo solo descubrirías cuando el programa ya está corriendo, posiblemente en las manos de un usuario.

Los programas con los que interactuamos todos los días trabajan con muchos tipos diferentes de información. Una aplicación de mensajería maneja textos, una calculadora maneja números, un videojuego maneja posiciones, velocidades, puntuaciones, y cientos de

otros datos. Para representar toda esta variedad, Zig ofrece varios tipos de datos fundamentales que puedes usar como bloques de construcción para representar cualquier tipo de información que tu programa necesite manejar.

Los tipos de datos más básicos en Zig se pueden agrupar en cuatro categorías principales: números enteros, números de punto flotante (decimales), booleanos (verdadero/falso), y caracteres/texto. Vamos a explorar cada uno de estos en detalle, comenzando por los números enteros, que son probablemente los que usarás con más frecuencia.

Los números enteros son exactamente lo que su nombre indica: números sin parte decimal. Ejemplos de enteros son 0, 1, 42, -17, 1000000. No incluyen fracciones ni decimales; son números "completos". Los computadores almacenan los enteros usando un número fijo de bits, y Zig te da control explícito sobre cuantos bits quieres usar para cada tipo de entero.

En Zig, los tipos de enteros se nombran de una manera muy descriptiva. El nombre indica si el número puede ser negativo (signed, con signo) o solo positivo (unsigned, sin signo), y cuantos bits de memoria usa. Por ejemplo, "i32" significa un entero con signo de 32 bits, mientras que "u8" significa un entero sin signo de 8 bits. La "i" viene de "integer" (con signo) y la "u" viene de "unsigned" (sin signo).

La cantidad de bits determina el rango de valores que el tipo puede representar. Un tipo de 8 bits puede representar 256 valores diferentes (2 elevado a 8). Si es un entero sin signo (u8), puede representar del 0 al 255. Si es un entero con signo (i8), puede representar del -128 al 127. De manera similar, un i32 (32 bits con signo) puede representar valores desde aproximadamente -2,147 millones hasta +2,147 millones, que es suficiente para la mayoría de los usos cotidianos.

Aquí están los tipos de enteros más comunes en Zig:

1	<code>const a: i8 = -100; // Entero con signo de 8 bits (-128 a 127)</code>
2	<code>const b: u8 = 200; // Entero sin signo de 8 bits (0 a 255)</code>
3	<code>const c: i16 = -30000; // Entero con signo de 16 bits</code>
4	<code>const d: u16 = 60000; // Entero sin signo de 16 bits</code>
5	<code>const e: i32 = -2000000000; // Entero con signo de 32 bits</code>
6	<code>const f: u32 = 4000000000; // Entero sin signo de 32 bits</code>
7	<code>const g: i64 = -9000000000000000000; // Entero con signo de 64 bits</code>
8	<code>const h: u64 = 18000000000000000000; // Entero sin signo de 64 bits</code>

Tipo	Bits	Rango	Uso Recomendado
u8	8	0 a 255	Bytes, caracteres ASCII
i8	8	-128 a 127	Valores pequeños con signo
u16	16	0 a 65,535	Contadores medianos
i16	16	-32,768 a 32,767	Valores medianos con signo
u32	32	0 a ~4 mil millones	índices, tamaños
i32	32	±2 mil millones	Propósito general (recomendado)
u64	64	0 a ~18 quintillones	Grandes cantidades
i64	64	±9 quintillones	Valores muy grandes con signo
usize	32/64	Depende de CPU	índices de arrays, tamaños (¡Muy común!)

Tabla 7 - Tipos de Enteros en Zig
(Fuente: Propia)

Observa que cada declaración incluye el tipo después de los dos puntos (por ejemplo, "i32"). Esto le indica explícitamente a Zig que tipo de datos estamos almacenando. Sin embargo, en muchos casos Zig puede inferir el tipo automáticamente basándose en el valor que estás asignando.

Cuando escribes "const x = 42;" sin especificar un tipo, Zig asume que quieres un entero de tamaño "comptime_int", que durante la compilación se convierte al tipo apropiado según cómo lo uses. comptime_int es un tipo especial que solo existe durante la compilación; el compilador lo convierte al tipo entero concreto apropiado según cómo se use la constante

¿Cuándo deberías usar cada tipo de entero? La respuesta depende de lo que estés representando. Si estás contando algo que nunca puede ser negativo (como la cantidad de usuarios o el tamaño de un archivo), usa un tipo sin signo (u8, u16, u32, etc.). Si el valor puede ser negativo (como una temperatura que puede bajar de cero o una posición relativa que puede ser hacia la izquierda o derecha), usa un tipo con signo (i8, i16, i32, etc.). En cuanto al tamaño, usa el tipo más pequeño que pueda contener todos los valores posibles que necesites representar, pero en caso de duda, i32 es una opción segura para la mayoría de los casos.



Consejo

Un error común de principiantes es no considerar el rango de los tipos de enteros. Si almacenas un valor que es demasiado grande o demasiado pequeño para el tipo que elegiste, Zig te dará un error de compilación en lugar de simplemente truncar el valor silenciosamente como hacen otros lenguajes. Esto es una característica de seguridad que te protege de errores sutiles.

Los números de punto flotante, o simplemente "flotantes", son números que pueden tener una parte decimal. Se llaman "de punto flotante" por la forma en que las computadoras los almacenan internamente: el punto decimal puede "flotar" hacia la izquierda o la derecha para representar números muy grandes o muy pequeños. Ejemplos de números flotantes son 3.14159, -0.001, 42.0, y 6.022e23 (notación científica para 6.022×10^{23}).

Zig ofrece dos tipos principales de punto flotante: f32 y f64. El f32 usa 32 bits y tiene aproximadamente 7 dígitos decimales de precisión. El f64 usa 64 bits y tiene aproximadamente 15 dígitos decimales de precisión.

1	<code>const pi: f64 = 3.14159265358979;</code>
2	<code>const temperatura: f32 = 36.5;</code>
3	<code>const muy_pequeño: f64 = 0.000000001;</code>
4	<code>const muy_grande: f64 = 1.5e10; // $1.5 \times 10^{10} = 15,000,000,000$</code>

Tipo	Bits	Precisión	Uso
f32	32	~7 dígitos	Gráficos, cuando la memoria es limitada
f64	64	~15 dígitos	Propósito general (recomendado)

Tabla 8 - Tipos de Punto Flotante
(Fuente: Propia)

Es importante entender una limitación fundamental de los números de punto flotante: no pueden representar todos los números decimales con precisión perfecta. Esto no es un defecto de Zig, sino una consecuencia de cómo funcionan los números binarios. Por ejemplo, el número decimal 0.1 no tiene una representación exacta en binario, igual que 1/3 no tiene una representación exacta en decimal (0.333...). Esto puede llevar a resultados sorprendentes en cálculos que parecen simples.

Por esta razón, nunca deberías comparar flotantes para ver si son exactamente iguales. En lugar de preguntar "¿es x igual a y?", deberías preguntar "¿es x suficientemente cercano a y?". Esta es una consideración importante que afecta a todos los lenguajes de programación, no solo a Zig. Para cálculos financieros donde la precisión exacta es crucial, muchos programadores prefieren usar enteros que representan centavos en lugar de flotantes que representan dólares.

Los tipos de punto flotante son perfectos para cálculos científicos, gráficos, físicas de juegos, y cualquier situación donde los valores son inherentemente aproximados. Evítalos para dinero, conteos exactos, o cualquier situación donde la precisión exacta sea crítica.

NOTA

El tipo booleano es el más simple de todos los tipos de datos: solo puede tener dos valores posibles: verdadero (true) o falso (false). A pesar de su simplicidad, los booleanos son increíblemente poderosos porque permiten que tu programa tome decisiones. "¿Está el usuario registrado?" "¿Es el número mayor que 100?" "¿Terminó el juego?" Todas estas preguntas tienen respuestas booleanas: sí o no, verdadero o falso.

En Zig, el tipo booleano se llama "bool", y sus únicos valores posibles son las palabras clave "true" y "false":

1	<code>const esta_lloviendo: bool = true;</code>
2	<code>const es_fin_de_semana: bool = false;</code>
3	<code>const usuario_activo = true; // Zig infiere que es bool</code>

Los booleanos suelen ser el resultado de comparaciones. Cuando comparas dos valores usando operadores como `==` (igual), `!=` (diferente), `<` (menor que), `>` (mayor que), el resultado es un booleano. Por ejemplo, `"5 > 3"` evalúa a `true` porque 5 sí es mayor que 3, mientras que `"10 == 20"` evalúa a `false` porque 10 no es igual a 20. Exploraremos estas comparaciones en detalle en el próximo capítulo sobre operadores.

También puedes combinar booleanos usando operadores lógicos: `"and"` (y), `"or"` (o), y `"!"` (no/negación). Por ejemplo, `"esta_lloviendo and es_fin_de_semana"` sería `true` solo si ambas condiciones son verdaderas. `"esta_lloviendo or es_fin_de_semana"` sería `true` si al menos una de las dos condiciones es verdadera. `"!esta_lloviendo"` invierte el valor: si `esta_lloviendo` es `true`, `"!esta_lloviendo"` es `false`, y viceversa.

Además de números y booleanos, los programas frecuentemente necesitan trabajar con texto. Zig maneja el texto usando dos conceptos relacionados pero distintos: caracteres individuales y cadenas de caracteres (strings). Entender la diferencia es importante para trabajar efectivamente con texto.

1	<code>const letra: u8 = 'A';</code>
2	<code>const digito: u8 = '7';</code>
3	<code>const símbolo: u8 = '@';</code>
4	<code>const espacio: u8 = ' ';</code>

Los strings (cadenas de caracteres) son secuencias de caracteres que forman texto. `"Hola, Mundo!"` es un string. `"42"` también es un string (aunque se vea como un número, las comillas lo hacen texto). En Zig, los strings se escriben entre comillas dobles y son de tipo `"[]const u8"`, que significa "un slice de bytes constantes".

1	<code>const saludo = "¡Hola, Mundo!";</code>
2	<code>const nombre = "Maria Garcia";</code>
3	<code>const mensaje = "Bienvenido al curso de Zig";</code>

Hay varios caracteres especiales que no puedes escribir directamente en un string porque tienen significados especiales. El más común es el salto de línea, que se representa como `"\n"`. La barra invertida (`\`) se llama "carácter de escape" y le dice a Zig que el siguiente carácter tiene un significado especial. Aquí están los caracteres de escape más comunes:

1	<code>const con_salto = "Primera línea\nSegunda línea";</code>
2	<code>const con_tab = "Columna1\tColumna2"; // \t es tabulador</code>
3	<code>const con_comilla = "Ella dijo \"Hola\""; // \" es comilla dentro de string</code>
4	<code>const con_barra = "C:\\Users\\nombre"; // \\ es una barra invertida</code>

Secuencia	Nombre	Resultado
<code>\\n</code>	Salto de línea	Nueva línea
<code>\\t</code>	Tabulador	Espacio de tabulación
<code>\\\"</code>	Comilla doble	Comilla dentro del string
<code>\\\\</code>	Barra invertida	Una barra \\

Tabla 9 - Caracteres de Escape Comunes
(Fuente: Propia)

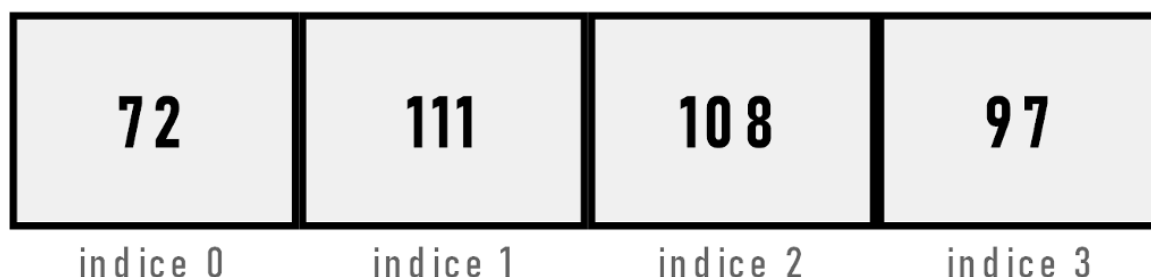


Figura 2 - Visualización del Array `hola_bytes`

Zig también soporta strings multilínea usando una sintaxis especial con dobles barras invertidas al principio de cada línea. Esto es útil cuando tienes texto largo que quieres dividir en varias líneas en tu código fuente sin usar `"\n"` manualmente.

Ya hemos mencionado que Zig puede inferir tipos automáticamente en muchos casos. Esto significa que no siempre necesitas especificar el tipo explícitamente; el compilador puede determinarlo basándose en el valor que asignas. Esta característica hace el código más conciso sin sacrificar seguridad.

1	<code>const x = 42;</code>	<code>// Zig infiere el tipo basado en el uso</code>
2	<code>const y = 3.14159;</code>	<code>// Inferido como flotante</code>
3	<code>const z = true;</code>	<code>// Inferido como bool</code>

Sin embargo, hay situaciones donde especificar el tipo explícitamente es necesario o deseable. Si quieres un tipo específico que no es el que Zig inferiría naturalmente (por ejemplo, quieres un `u8` en lugar de un entero más grande), debes especificarlo. También es buena práctica especificar tipos explícitamente en las firmas de funciones y en código que otros van a leer, ya que hace las intenciones más claras.

La regla general es: si el tipo es obvio por el contexto, puedes dejar que Zig lo infiera. Si hay alguna ambigüedad o si el tipo específico es importante para el funcionamiento correcto del programa, especifícalo explícitamente. Con el tiempo, desarrollarás intuición sobre cuando cada enfoque es más apropiado.

A veces necesitas declarar una variable, pero no tienes un valor para asignarle todavía. En Zig, puedes usar el valor especial `"undefined"` para esto. Sin embargo, `"undefined"` es diferente de `null` en otros lenguajes: significa que la variable no tiene un valor válido todavía y acceder a ella sería un error. Zig usa `undefined` para detectar errores donde intentas usar valores no inicializados.

1	<pre>var resultado: i32 = undefined; // Declarado, pero no inicializado</pre>
2	<pre>// ... más código que eventualmente asigna un valor a resultado ...</pre>
3	<pre>resultado = 100; // Ahora tiene un valor valido</pre>



NOTA

ADVERTENCIA: `'undefined'` no es lo mismo que `'null'` o `'vacío'`. Significa que la variable contiene basura aleatoria de la memoria. Si intentas leer una variable `'undefined'` antes de escribirle un valor, tu programa puede comportarse de manera impredecible. Úsalo solo cuando estés seguro de que le darás un valor inmediatamente.

Zig también tiene tipos "opcionales" que pueden contener un valor o pueden estar vacíos de manera segura. Esto es diferente de undefined: un opcional vacío es un estado válido que puedes verificar, mientras que undefined es un estado inválido que indica que olvidaste inicializar algo. Los opcionales se indican con un signo de interrogación antes del tipo:

1	<code>var maybe_number: ?i32 = null; // Opcional que actualmente está vacío</code>
2	<code>maybe_number = 42; // Ahora contiene 42</code>
3	<code>maybe_number = null; // Vuelve a estar vacío</code>

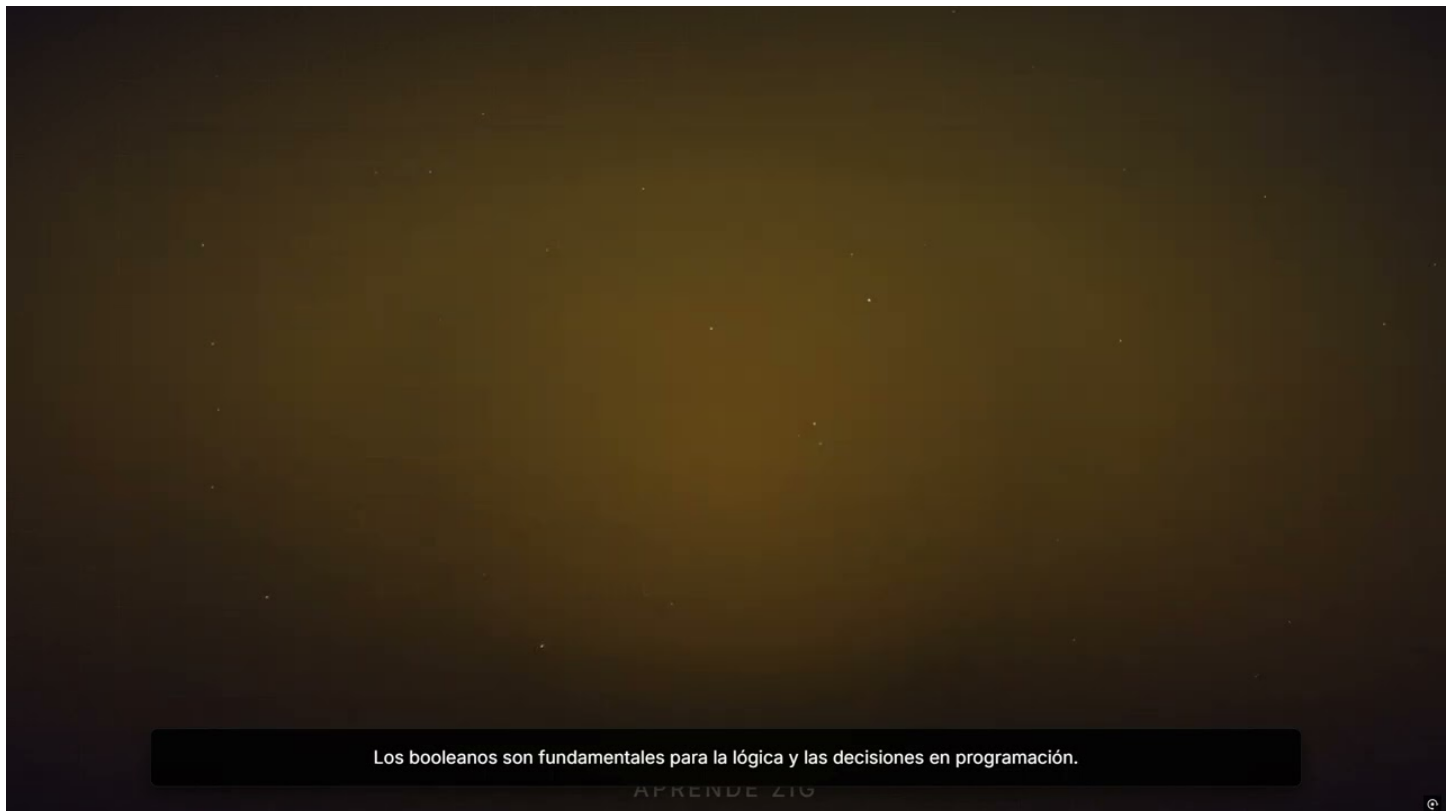
Los opcionales son extremadamente útiles para representar valores que podrían no existir: el resultado de buscar algo que quizás no se encuentra, un campo de formulario que el usuario podría dejar en blanco, o una configuración que tiene un valor por defecto si no se especifica. Aprenderemos más sobre cómo trabajar con opcionales cuando veamos las estructuras de control.

Veamos un programa completo que demuestra varios de los conceptos que hemos aprendido en este capítulo. Este programa declara diferentes tipos de variables y las muestra en pantalla:

1	<code>const std = @import("std");</code>
2	
3	<code>pub fn main() void {</code>
4	<code> // Constantes de diferentes tipos</code>
5	<code> const nombre = "Carlos";</code>
6	<code> const edad: u8 = 28;</code>
7	<code> const altura: f64 = 1.75;</code>
8	<code> const es_estudiante: bool = true;</code>
9	
10	<code> // Variable que puede cambiar</code>
11	<code> var contador: i32 = 0;</code>
12	<code> contador = contador + 1;</code>
13	
14	<code> // Mostrando los valores</code>

15	<code>std.debug.print("Nombre: {s}\n", .{nombre});</code>
16	<code>std.debug.print("Edad: {}\n", .{edad});</code>
17	<code>std.debug.print("Altura: {d:.2}\n", .{altura});</code>
18	<code>std.debug.print("¿Es estudiante? {}\n", .{es_estudiante});</code>
19	<code>std.debug.print("Contador: {}\n", .{contador});</code>
20	<code>}</code>

Video 3 - Variables y tipos de datos en Zig



Este programa introduce algo nuevo: cómo mostrar variables usando `std.debug.print`. Los símbolos entre llaves `{}` son marcadores de posición que se reemplazan con los valores que pasamos en la lista `{...}`. El `{s}` se usa para strings, `{}` para la mayoría de los otros tipos, y `{d:.2}` para flotantes con 2 decimales.

En este capítulo hemos explorado uno de los conceptos más fundamentales de la programación: las variables y los tipos de datos. Aprendiste que las variables son como contenedores con nombres que almacenan valores, y que Zig distingue entre constantes (`const`), cuyos valores no pueden cambiar, y variables mutables (`var`), cuyos valores sí pueden modificarse.

Exploramos los tipos de datos básicos de Zig: los enteros con signo (i8, i16, i32, i64) y sin signo (u8, u16, u32, u64) para números enteros; los tipos de punto flotante (f32, f64) para números decimales; el tipo bool para valores verdadero/falso; y los caracteres y strings para trabajar con texto. Cada tipo tiene su propósito específico, y elegir el tipo correcto para cada situación es una habilidad importante.

En el próximo capítulo aprenderemos sobre los operadores: las herramientas que te permiten manipular y combinar valores. Veremos operadores aritméticos para hacer cálculos matemáticos, operadores de comparación para comparar valores, y operadores lógicos para combinar condiciones. Con estos conocimientos, podrás empezar a escribir programas que hacen cálculos y toman decisiones.

EJERCICIOS

Pon a prueba lo aprendido con los siguientes ejercicios:

DECLARANDO VARIABLES BÁSICAS

Crea un programa que declare las siguientes variables con los tipos y valores apropiados: tu nombre (como string), tu año de nacimiento (como entero), tu estatura en metros (como flotante), y si tienes mascotas (como booleano). Luego, muestra cada valor en pantalla con un texto descriptivo.

Ejercicio

CONST VS VAR

Escribe un programa que demuestre la diferencia entre const y var. Declara una constante llamada 'valor_fijo' con el valor 100, y una variable llamada 'valor_variable' con el valor inicial 0. Intenta reasignar ambas a un nuevo valor. ¿Qué sucede cuando intentas compilar el programa? Comenta la línea que causa el error y explica por qué falla.

Ejercicio

TIPOS DE ENTEROS

Experimenta con los límites de los tipos de enteros. Intenta declarar: (a) un u8 con el valor 300, (b) un i8 con el valor 200, (c) un u8 con el valor -5. Compila el programa después de cada intento. ¿Qué errores muestra Zig? ¿Por qué ocurren estos errores?

Ejercicio

3

CALCULANDO EL ÁREA

Escribe un programa que calcule y muestre el área de un rectángulo. Declara dos constantes para el ancho (7.5) y el alto (4.2), calcula el área multiplicándolos, guarda el resultado en una tercera constante, y muestra los tres valores con mensajes descriptivos. Usa el tipo f64 para todos los

Ejercicio

4

CARÁCTERES ESPECIALES

Crea un programa que muestre un pequeño 'recibo' en la terminal usando caracteres especiales. El recibo debe tener: una línea de guiones como borde superior, el nombre de una tienda, una línea en blanco, tres productos con precios separados por tabuladores, otra línea de guiones, y un total. Usa `\n` para saltos de línea y `\t` para tabuladores. si el editor lo subraya en rojo antes de compilar. Anota qué funcionalidades observas.

Ejercicio

5

INFERENCIA Y CONVERSIÓN DE TIPOS

Experimenta con la inferencia de tipos y las conversiones: (a) Declara una constante 'número' sin especificar tipo y asígnale el valor 42. Luego intenta usarla como `u8` en un `print`. (b) Declara un `i32` con valor 1000 e intenta asignarlo a una variable `u8`. ¿Qué error ocurre? (c) Usa `@intCast` para convertir un `i32` a `u8` con un valor que si quepa (por ejemplo, 100). (d) ¿Qué pasa si intentas convertir un valor que no cabe (por ejemplo, 300)? muestra y explica que te dice.

Ejercicio

6

CAPÍTULO 4

Operadores y expresiones

Ahora que sabes cómo almacenar información en variables, es hora de aprender a manipularla. Los operadores son símbolos especiales que le indican a Zig que realice operaciones sobre uno o más valores. Así como en matemáticas usas el signo más (+) para sumar números, en programación usas operadores para realizar cálculos, comparar valores, combinar condiciones lógicas, y mucho más. Las expresiones son combinaciones de valores, variables y operadores que producen un resultado. Por ejemplo, "5 + 3" es una expresión que usa el operador de suma y produce el resultado 8.

Dominar los operadores es fundamental para cualquier programador porque son las herramientas básicas con las que manipulas datos. Sin ellos, tus programas solo podrían almacenar información estática sin poder hacer nada útil con ella. Con los operadores, puedes calcular totales, determinar si un usuario cumple ciertos requisitos, decidir qué camino debe seguir tu programa, y construir lógica compleja a partir de piezas simples.

Zig proporciona varios tipos de operadores, cada uno diseñado para un propósito específico. Los operadores aritméticos realizan cálculos matemáticos como suma, resta y multiplicación. Los operadores de comparación comparan valores y producen resultados booleanos (verdadero o falso). Los operadores lógicos combinan o modifican valores booleanos. Los operadores de asignación asignan valores a variables. Y los operadores de bits manipulan números a nivel binario. Exploraremos cada categoría en detalle, comenzando con los más familiares: los operadores aritméticos.

1	const a: u32 = 10;
2	const b: u32 = 3;
3	
4	const suma = a + b; // 13
5	const resta = a - b; // 7
6	const producto = a * b; // 30
7	const división = a / b; // 3 (división entera)
8	const módulo = a % b; // 1 (residuo de 10 ÷ 3)

Operador	Nombre	Ejemplo	Resultado
+	Suma	10+3	13
-	Resta	10-3	7
*	Multiplicación	10*3	30
/	División	10/3	3(entera)
%	Módulo	10%3	1(residuo)

Tabla 10 - Operadores Aritméticos
(Fuente: Propia)

La suma, resta y multiplicación funcionan exactamente como esperarías. El operador de suma (+) toma dos números y produce su suma. El operador de resta (-) produce la diferencia entre el primer número y el segundo. El operador de multiplicación (*) produce el producto de ambos números. No hay sorpresas aquí; estos operadores se comportan de la manera intuitiva.

La división, sin embargo, tiene una característica importante que debes entender. Cuando divides dos números enteros en Zig, el resultado es también un número entero, lo que significa que la parte decimal se descarta. Esto se llama "división entera" o "división truncada". En el ejemplo anterior, 10 dividido entre 3 matemáticamente es 3.333..., pero como estamos trabajando con enteros, Zig descarta el .333 y nos da simplemente 3. Si necesitas el resultado decimal, debes usar números de punto flotante en lugar de enteros.

1	<code>const x: f64 = 10.0;</code>
2	<code>const y: f64 = 3.0;</code>
3	<code>const división_decimal = x / y; // 3.333333...</code>

El operador módulo (%) es quizás nuevo para ti. Este operador calcula el residuo de una división. Cuando divides 10 entre 3, obtienes 3 con un residuo de 1 (porque $3 \times 3 = 9$, y $10 - 9 = 1$). El operador módulo te da precisamente ese residuo. Aunque puede parecer oscuro al principio, el módulo es extremadamente útil en programación. Por ejemplo, puedes usarlo para determinar si un número es par o impar: si `número % 2` es igual a 0, el número es par; si es igual a 1, el número es impar.

1	<code>const número = 17;</code>
2	<code>const residuo = número % 2; // 1, por lo tanto 17 es impar</code>
3	
4	<code>// Ejemplos prácticos del módulo:</code>
5	<code>const horas = 25 % 24; // 1 (25 horas = 1 día y 1 hora)</code>
6	<code>const dia_semana = 10 % 7; // 3 (el día 10 es el 3er día de la 2da semana)</code>

También existe el operador de negación unario (-), que convierte un número positivo en negativo o viceversa. A diferencia de los otros operadores aritméticos que trabajan con dos valores (son "binarios"), la negación solo necesita un valor (es "unario"). Simplemente colocas el signo menos antes de un número o variable para negarlo.

1	<code>const positivo: i32 = 42;</code>
2	<code>const negativo = -positivo; // -42</code>
3	<code>const doble_negacion = -negativo; // 42 (vuelve a ser positivo)</code>

Zig tiene protección contra desbordamiento aritmético. Si una operación produce un resultado que es demasiado grande o pequeño para el tipo de datos, Zig detectará esto y generará un error en lugar de producir un resultado incorrecto silenciosamente. Esta es una característica de seguridad importante que te protege de errores sutiles.

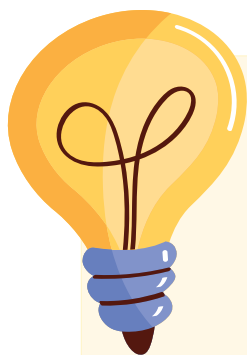
NOTA

Cuando una expresión contiene múltiples operadores, Zig necesita saber en qué orden evaluarlos. Esto se llama "precedencia de operadores" y funciona de manera similar a las reglas matemáticas que ya conoces. La multiplicación y división tienen mayor precedencia que la suma y resta, lo que significa que se evalúan primero.

1	<code>const resultado1 = 2 + 3 * 4; // 14, no 20 (primero 3*4=12, luego 2+12=14)</code>
2	<code>const resultado2 = (2 + 3) * 4; // 20 (los paréntesis fuerzan 2+3 primero)</code>

En el primer ejemplo, aunque la suma aparece antes en la expresión, la multiplicación se evalúa primero porque tiene mayor precedencia.

Cuando tienes operadores de la misma precedencia, se evalúan de izquierda a derecha. Por ejemplo, en "10 - 3 - 2", primero se calcula 10 - 3 = 7, y luego 7 - 2 = 5. Esto se llama "asociatividad izquierda". Mi recomendación práctica es: cuando tengas dudas sobre el orden de las operaciones, usa paréntesis. Incluso si no son estrictamente necesarios, los paréntesis hacen el código más claro y evitan errores causados por malentendidos sobre la precedencia.



Consejo

No intentes memorizar todas las reglas de precedencia. Es mucho más práctico usar paréntesis siempre que haya cualquier posibilidad de ambigüedad. El código con paréntesis explícitos es más fácil de leer y menos propenso a errores. Escribe $(a + b) * c$ en lugar de depender de saber qué $*$ tiene mayor precedencia que $+$.

Los operadores de comparación comparan dos valores y producen un resultado booleano: verdadero (true) si la comparación es cierta, falso (false) si no lo es. Estos operadores son esenciales para tomar decisiones en tu programa. ¿Es el usuario mayor de edad? ¿Tiene suficiente dinero en su cuenta? ¿Ya llegamos al final de la lista? Todas estas preguntas se responden usando operadores de comparación.

Zig proporciona seis operadores de comparación: igual a (`==`), diferente de (`!=`), menor que (`<`), mayor que (`>`), menor o igual que (`<=`), y mayor o igual que (`>=`). Cada uno compara los valores a su izquierda y derecha y produce un resultado booleano.

1	<code>const a: i32 = 10;</code>	
2	<code>const b: i32 = 20;</code>	
3	<code>const c: i32 = 10;</code>	
4		
5	<code>const igual = a == c;</code>	<code>// true (10 es igual a 10)</code>
6	<code>const diferente = a != b;</code>	<code>// true (10 es diferente de 20)</code>
7	<code>const menor = a < b;</code>	<code>// true (10 es menor que 20)</code>
8	<code>const mayor = a > b;</code>	<code>// false (10 no es mayor que 20)</code>
9	<code>const menor_igual = a <= c;</code>	<code>// true (10 es menor o igual a 10)</code>
10	<code>const mayor_igual = b >= a;</code>	<code>// true (20 es mayor o igual a 10)</code>

Operador	Nombre	Ejemplo	Resultado
==	Igual a	10 == 10	true
!=	Diferente de	10 != 20	true
<	Menor que	10 < 20	true
>	Mayor que	10 > 20	false
<=	Menor o igual	10 <= 10	true
>=	Mayor o igual	20 >= 10	true

Tabla 11 - Operadores de Comparación
(Fuente: Propia)

Un error muy común para principiantes es confundir el operador de asignación (=) con el operador de igualdad (==). El signo igual simple (=) asigna un valor a una variable. El doble signo igual (==) compara dos valores. Esta distinción es crucial. Si escribes "if (x = 5)" cuando querías escribir "if (x == 5)", tu programa no hará lo que esperas. Zig, afortunadamente, te protege de algunos de estos errores porque no permite asignaciones dentro de expresiones condicionales.

Los operadores de comparación funcionan con diferentes tipos de datos. Puedes comparar enteros, flotantes, caracteres, e incluso booleanos. Sin embargo, debes tener cuidado al comparar flotantes para igualdad exacta debido a los problemas de precisión que mencionamos en el capítulo anterior. En general, es mejor comparar si dos flotantes están "suficientemente cerca" en lugar de si son exactamente iguales.

1	<code>const letra_a: u8 = 'A'; // 65 en ASCII</code>
2	<code>const letra_b: u8 = 'B'; // 66 en ASCII</code>
3	<code>const comparacion = letra_a < letra_b; // true ('A' viene antes que 'B')</code>

Cuando comparas caracteres, estás comparando sus valores numéricos en la tabla ASCII o Unicode. Las letras mayúsculas tienen valores menores que las minúsculas (A=65, a=97), y las letras están ordenadas alfabéticamente dentro de su grupo. Esto significa que 'A' < 'B' es verdadero, y 'Z' < 'a' también es verdadero (porque las mayúsculas tienen valores menores que las minúsculas).

Los operadores lógicos trabajan con valores booleanos y te permiten combinar múltiples condiciones. Hay tres operadores lógicos principales: AND (y), OR (o), y NOT (no). En Zig, estos se escriben como "and", "or", y "!" respectivamente.

El operador AND (and) produce verdadero solo si ambas condiciones son verdaderas. Piensa en él como "la primera condición Y la segunda condición deben ser ciertas".

1	<code>const tiene_dinero = true;</code>
2	<code>const tiene_tiempo = true;</code>
3	<code>const puede_viajar = tiene_dinero and tiene_tiempo; // true</code>
4	
5	<code>const llueve = true;</code>
6	<code>const tiene_paraguas = false;</code>
7	<code>const sale_sin_mojarse = llueve and tiene_paraguas; // false</code>

El operador OR (or) produce verdadero si al menos una de las condiciones es verdadera. Piensa en él como "la primera condición O la segunda condición debe ser cierta". Solo produce falso si ambas condiciones son falsas.

1	<code>const tiene_efectivo = false;</code>
2	<code>const tiene_tarjeta = true;</code>
3	<code>const puede_pagar = tiene_efectivo or tiene_tarjeta; // true</code>
4	
5	<code>const es_festivo = false;</code>
6	<code>const es_fin_semana = false;</code>
7	<code>const descansa = es_festivo or es_fin_semana; // false</code>

El operador NOT (!) invierte un valor booleano. Si el valor es verdadero, NOT lo convierte en falso. Si el valor es falso, NOT lo convierte en verdadero. Este operador es unario, lo que significa que solo necesita un operando.

1	<code>const esta_lloviendo = true;</code>
2	<code>const no_esta_lloviendo = !esta_lloviendo; // false</code>
3	
4	<code>const puerta_cerrada = false;</code>
5	<code>const puerta_abierta = !puerta_cerrada; // true</code>

Operador	Nombre	Descripción	Ejemplo
And	Y lógico	True si ambos son true	True and false
Or	O Lógico	True si al menos uno es true	True or false
!	Negación	Invierte el valor	!true → false

Tabla 12 - Operadores Lógicos
(Fuente: Propia)

Puedes combinar múltiples operadores lógicos para crear expresiones complejas. Cuando lo hagas, recuerda que NOT tiene la mayor precedencia, seguido de AND, y luego OR. Esto significa que "a or b and c" se evalúa como "a or (b and c)", no como "(a or b) and c". Como siempre, cuando haya posibilidad de confusión, usa paréntesis para hacer explícito el orden de evaluación.

1	<code>const es_adulto = true;</code>
2	<code>const tiene_identificacion = true;</code>
3	<code>const es_vip = false;</code>
4	
5	<code>// Puede entrar si: es adulto CON identificacion, O si es VIP</code>
6	<code>const puede_entrar = (es_adulto and tiene_identificacion) or es_vip; // true</code>



Zig usa 'evaluación de cortocircuito' para los operadores lógicos. Esto significa que, si el resultado ya puede determinarse solo con el primer operando, el segundo no se evalúa. Por ejemplo, en 'false and x', el resultado siempre será false sin importar el valor de x, así que x ni siquiera se evalúa. Esto puede ser útil para evitar errores cuando el segundo operando podría causar problemas.

El operador de asignación básico (=) ya lo conoces: asigna el valor de la derecha a la variable de la izquierda. Pero Zig también proporciona operadores de asignación compuestos que combinan una operación aritmética con una asignación. Estos operadores son una forma abreviada de escribir operaciones comunes.

1	var contador: u32 = 10;	
2		
3	contador = contador + 5; // Forma larga: ahora contador es 15	
4	contador += 5; // Forma abreviada equivalente: ahora es 20	
5		
6	contador -= 3; // Equivale a: contador = contador - 3 (ahora es 17)	
7	contador *= 2; // Equivale a: contador = contador * 2 (ahora es 34)	
8	contador /= 2; // Equivale a: contador = contador / 2 (ahora es 17)	
9	contador %= 5; // Equivale a: contador = contador % 5 (ahora es 2)	

Operador	Equivale a	Ejemplo
+-	x=x+n	Contador +=5
-=	x=x-n	Contador-=3
*=	x=x*n	Contador *=2
/=	x=x/n	contador /=2
%=	x=x%n	Contador %=5

Tabla 13 - Operadores de Asignación Compuestos
(Fuente: Propia)

Estos operadores compuestos (+=, -=, *=, /=, %=) son muy utilizados porque hacen el código más conciso y más fácil de leer. En lugar de escribir "contador = contador + 1" cada vez que quieras incrementar una variable, puedes simplemente escribir "contador += 1". El ahorro en escritura puede parecer menor, pero cuando escribes código regularmente, estas pequeñas mejoras se acumulan.

Es importante recordar que solo puedes usar estos operadores con variables declaradas con "var", no con constantes declaradas con "const". Si intentas usar un operador de

asignación con una constante, el compilador te dará un error porque estarías intentando modificar un valor que se declaró como inmutable.

Los operadores de bits (también llamados operadores bitwise) manipulan números a nivel de sus representaciones binarias individuales. Cada número en una computadora se almacena como una secuencia de bits (0s y 1s), y estos operadores te permiten manipular esos bits directamente. Aunque esto puede parecer esotérico, los operadores de bits son muy útiles en programación de sistemas, juegos, criptografía, y cualquier situación donde necesites optimizar el uso de memoria o la velocidad.

Los operadores de bits principales son: AND (&), OR (|), XOR (^), NOT (~), desplazamiento a la izquierda (<<), y desplazamiento a la derecha (>>). Veamos cada uno brevemente.

1	<code>const a: u8 = 0b11001100; // 204 en decimal</code>
2	<code>const b: u8 = 0b10101010; // 170 en decimal</code>
3	
4	<code>const and_result = a & b; // 0b10001000 = 136 (bit 1 donde ambos tienen 1)</code>
5	<code>const or_result = a b; // 0b11101110 = 238 (bit 1 donde cualquiera tiene 1)</code>
6	<code>const xor_result = a ^ b; // 0b01100110 = 102 (bit 1 donde solo uno tiene 1)</code>

Operador	Nombre	Descripción
&	AND	1 solo si ambos bits son 1
	OR	1 si cualquier bit es 1
^	XOR	1 si exactamente un bit es 1
~	NOT	Invierte todos los bits
<<	Desp. Izquierda	Multiplica por potencia de 2
>>	Desp. Derecha	Divide por potencia de 2

Tabla 14 - Operadores de Bits
(Fuente: Propia)



Figura 3 - Operaciones bitwise

El operador AND a nivel de bits (&) compara cada par de bits correspondientes. Si ambos son 1, el resultado es 1; si cualquiera es 0, el resultado es 0. El operador OR (|) produce 1 si cualquiera de los bits es 1. El operador XOR (^) produce 1 solo si exactamente uno de los bits es 1, pero no ambos. El NOT (~) invierte todos los bits.

Los operadores de desplazamiento mueven todos los bits hacia la izquierda o la derecha. Desplazar a la izquierda es equivalente a multiplicar por potencias de 2, y desplazar a la derecha es equivalente a dividir por potencias de 2. Estas operaciones son extremadamente rápidas porque las computadoras están optimizadas para realizarlas.

1	const n: u8 = 0b00000100; // 4 en decimal
2	
3	const izq = n << 2; // 0b00010000 = 16 (4 × 2 ² = 4 × 4 = 16)
4	const der = n >> 1; // 0b00000010 = 2 (4 ÷ 2 ¹ = 4 ÷ 2 = 2)



Consejo

No te preocupes si los operadores de bits te parecen confusos ahora. Son una herramienta avanzada que no necesitarás para la mayoría de los programas básicos. A medida que avances en tu carrera como programador, encontrarás situaciones específicas donde son útiles, y entonces tendrá más sentido profundizar en ellos.

Zig ofrece una forma compacta de elegir entre dos valores basándose en una condición: las expresiones condicionales con "if". A diferencia de las sentencias if que veremos en el próximo capítulo, las expresiones if producen un valor que puedes asignar directamente a una variable.

1	<code>const edad: u8 = 20;</code>
2	<code>const categoría = if (edad ≥ 18) "adulto" else "menor";</code>
3	
4	<code>const temperatura: i8 = -5;</code>
5	<code>const estado = if (temperatura > 0) "caliente" else if (temperatura == 0) "congelando" else "helado";</code>

En el primer ejemplo, la variable "categoría" se asigna con el valor "adulto" si la edad es mayor o igual a 18, o con "menor" si no lo es. Esta es una forma muy concisa de escribir lógica simple de selección. Puedes encadenar múltiples condiciones usando "else if" como se muestra en el segundo ejemplo.

Las expresiones condicionales son útiles cuando necesitas asignar uno de varios valores a una variable basándote en una condición. Son más concisas que usar sentencias if completas y declarar la variable por separado. Sin embargo, para lógica más compleja que involucra múltiples sentencias, es mejor usar las estructuras de control que veremos en el próximo capítulo.

A veces necesitas usar un valor de un tipo en un contexto que espera otro tipo. Por ejemplo, podrías tener un entero y necesitar un flotante, o un número pequeño que quieres almacenar en un tipo más grande. Zig es muy estricto sobre mezclar tipos, pero proporciona formas explícitas de convertir entre ellos usando la función @intCast, @floatCast, @intToFloat, y similares.

1	<code>const pequeño: u8 = 42;</code>
2	<code>const grande: u32 = pequeño; // OK: conversión implícita a tipo más grande</code>
3	
4	<code>const entero: i32 = 100;</code>
5	<code>const flotante: f64 = @floatFromInt(entero); // 100.0</code>
6	
7	<code>const decimal: f64 = 3.7;</code>
8	<code>const truncado: i32 = @intFromFloat(decimal); // 3 (se trunca el decimal)</code>

Zig permite la conversión implícita cuando es seguro hacerlo, como convertir un u8 a un u32 (el valor más pequeño siempre cabe en el tipo más grande). Sin embargo, requiere conversión explícita cuando podría haber pérdida de información, como convertir un entero grande a uno pequeño o un flotante a un entero. Esta explicitud te protege de errores sutiles donde los valores se truncan o cambian silenciosamente.

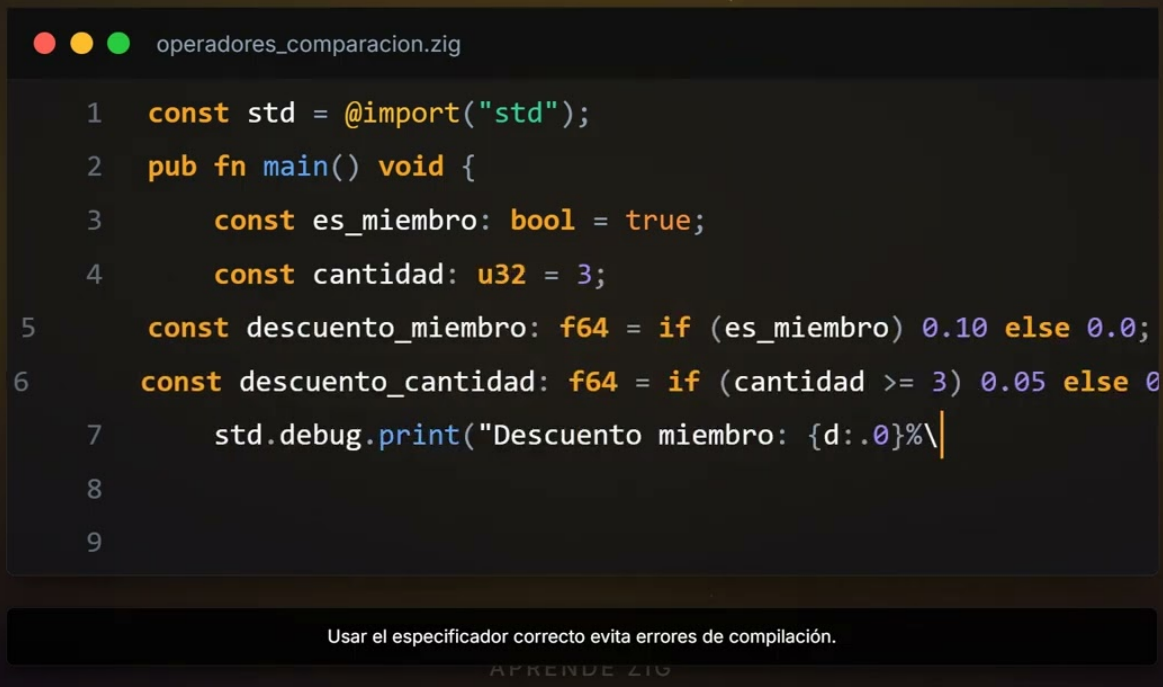
NOTA

Las funciones de conversión que empiezan con @ son funciones integradas (builtins) de Zig. Siempre empiezan con el símbolo @ y proporcionan operaciones fundamentales del lenguaje. Verás más de estas funciones a medida que avances en el libro.

Veamos un programa completo que utiliza varios de los operadores que hemos aprendido. Este programa simula una calculadora simple de descuentos que calcula el precio final de un producto aplicando diferentes reglas.

1	<code>const std = @import("std");</code>
2	
3	<code>pub fn main() void {</code>
4	<code> const precio_original: f64 = 100.0;</code>
5	<code> const es_miembro: bool = true;</code>
6	<code> const cantidad: u32 = 3;</code>
7	
8	<code> // Calcular descuento base (10% para miembros)</code>
9	<code> const descuento_miembro: f64 = if (es_miembro) 0.10 else 0.0;</code>
10	
11	<code> // Descuento adicional por cantidad (5% si compra 3 o más)</code>
12	<code> const descuento_cantidad: f64 = if (cantidad ≥ 3) 0.05 else 0.0;</code>
13	
14	<code> // Descuento total</code>
15	<code> const descuento_total = descuento_miembro + descuento_cantidad;</code>
16	
17	<code> // Precio final</code>
18	<code> const precio_final = precio_original * (1.0 - descuento_total);</code>
19	
20	<code> // Mostrar resultados</code>
21	<code> std.debug.print("Precio original: \${d:.2}\n", .{precio_original});</code>
22	<code> std.debug.print("Descuento aplicado: {d:.0}%\n", .{descuento_total * 100});</code>
23	<code> std.debug.print("Precio final: \${d:.2}\n", .{precio_final});</code>
24	<code>}</code>

Video 4 - Operadores y expresiones en Zig



```
1  const std = @import("std");
2  pub fn main() void {
3      const es_miembro: bool = true;
4      const cantidad: u32 = 3;
5      const descuento_miembro: f64 = if (es_miembro) 0.10 else 0.0;
6      const descuento_cantidad: f64 = if (cantidad >= 3) 0.05 else 0.0;
7      std.debug.print("Descuento miembro: {d:.0}%\n",
8                      .{descuento_miembro});
9  }
```

Usar el especificador correcto evita errores de compilación.

Este programa demuestra varios conceptos: usa operadores aritméticos para calcular el descuento y el precio final, operadores de comparación para determinar si se aplican los descuentos, y expresiones condicionales para seleccionar los valores apropiados. Aunque es un ejemplo simple, ilustra cómo los diferentes tipos de operadores trabajan juntos para crear lógica útil.

En este capítulo hemos explorado los operadores de Zig, las herramientas fundamentales para manipular datos en tus programas. Aprendiste sobre los operadores aritméticos (+, -, *, /, %) que te permiten realizar cálculos matemáticos, y como el operador módulo es útil para obtener residuos de divisiones.

Viste cómo la precedencia de operadores determina el orden de evaluación, y cómo los paréntesis te dan control explícito sobre este orden.

Exploramos los operadores de comparación (==, !=, <, >, <=, >=) que producen valores booleanos al comparar valores, y los operadores lógicos (and, or, !) que te permiten combinar condiciones booleanas para crear expresiones más complejas. También viste los operadores de asignación compuestos (+=, -=, etc.) que simplifican operaciones comunes de actualización de variables.

Aunque los operadores de bits pueden parecer avanzados, ahora tienes una idea de cómo funcionan para cuando los necesites en el futuro.

Finalmente, aprendiste sobre las expresiones condicionales y las conversiones de tipos, que te dan más flexibilidad al trabajar con diferentes tipos de datos.

En el próximo capítulo aprenderemos sobre las estructuras de control: las sentencias if, los bucles while y for, y cómo usarlas para hacer que tu programa tome decisiones y repita acciones. Con estas herramientas, podrás crear programas que respondan dinámicamente a diferentes condiciones y procesen grandes cantidades de datos.

EJERCICIOS

Pon a prueba tus conocimientos con los siguientes ejercicios:

CALCULADORA BÁSICA

Escribe un programa que declare dos constantes numéricas con los valores 15 y 4, y luego calcule y muestre: su suma, su diferencia, su producto, su división (tanto entera como decimal), y el residuo de su división. Asegúrate de usar tipos apropiados para cada operación.

Ejercicio

PAR O IMPAR

Escribe un programa que determine si un número es par o impar usando el operador módulo. Declara una constante con cualquier número entero y usa una expresión condicional para asignar el string "par" o "impar" a una variable, luego muéstralo.

Ejercicio

COMPARANDO EDADES

Escribe un programa que declare las edades de tres personas (por ejemplo: Ana tiene 25 años, Luis tiene 30, y María tiene 25). Usa operadores de comparación para determinar y mostrar: si Ana es mayor que Luis, si Ana y Maria tienen la misma edad, y quien es el mayor de los tres (puedes usar múltiples comparaciones)

Ejercicio

3

CONDICIONES COMBINADAS

Una persona puede obtener un préstamo si cumple AMBAS condiciones: tener al menos 18 años Y tener un ingreso mensual de al menos 1000. Escribe un programa que declare variables para la edad y el ingreso de una persona, use operadores lógicos para determinar si califica, y muestre el resultado.

Ejercicio

4

OPERADORES DE ASIGNACIÓN COMPUESTOS

Escribe un programa que simule una cuenta bancaria simple. Comienza con un saldo de 1000, luego realiza las siguientes operaciones usando operadores de asignación compuestos: deposita 500, retira 200, deposita 100, y muestra el saldo después de cada operación.

Ejercicio

5

CAPÍTULO 5

Estructuras de control

Hasta ahora, todos los programas que hemos escrito se ejecutan de arriba hacia abajo, una instrucción tras otra, sin desviarse. Pero los programas realmente útiles necesitan poder tomar decisiones y repetir acciones. ¿Debe el programa mostrar un mensaje de bienvenida o un mensaje de error? ¿Debe procesar un elemento o cien? ¿Debe continuar ejecutándose o terminar? Las estructuras de control son las herramientas que nos permiten responder estas preguntas, dándole a nuestros programas la capacidad de elegir caminos diferentes y de repetir acciones tantas veces como sea necesario.

Imagina que estás dando instrucciones para llegar a tu casa. No dirías simplemente "camina recto durante 500 metros". Dirías algo como "camina hasta llegar a la farmacia, luego gira a la izquierda. Si ves una fuente, pasaste mi casa; retrocede un poco". Las estructuras de control en programación funcionan de manera similar: permiten que tu programa tome decisiones basadas en condiciones y repita acciones hasta que se cumpla cierta condición.

En este capítulo aprenderemos sobre dos tipos fundamentales de estructuras de control. Las estructuras condicionales (if, else, switch) permiten que tu programa elija qué código ejecutar basándose en condiciones. Las estructuras de repetición (while, for) permiten que tu programa ejecute el mismo código múltiples veces. Juntas, estas estructuras te dan el poder de crear programas que responden dinámicamente a diferentes situaciones y qué pueden procesar grandes cantidades de datos.

Tipo	Estructura	Uso principal
Condicional	if / else	Ejecutar código según condición
Condicional	switch	Comparar contra múltiples valores
Repetición	while	Repetir mientras condición sea true
Repetición	For	Iterar sobre colecciones o rangos

Tabla 15 - Estructuras de Control en Zig
(Fuente: Propia)

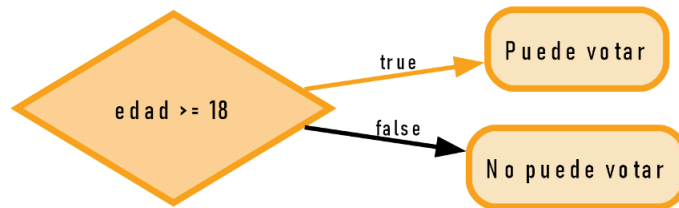


Figura 4 - Flujo de decisión if/else

La sentencia if es la estructura condicional más básica y probablemente la más usada en programación. Funciona exactamente como su nombre sugiere: Si una condición es verdadera, ejecuta un bloque de código. La sintaxis básica en Zig es:

1	<code>if (condición) {</code>
2	<code> // código que se ejecuta si la condición es verdadera</code>
3	<code>}</code>

La condición entre paréntesis debe ser una expresión que evalúe a un valor booleano (true o false). Si la condición es verdadera, se ejecuta todo el código dentro de las llaves. Si la condición es falsa, el programa simplemente salta ese bloque y continua con el código que viene después. Veamos un ejemplo concreto:

1	<code>const edad: u8 = 20;</code>
2	
3	<code>if (edad ≥ 18) {</code>
4	<code> std.debug.print("Eres mayor de edad\n", .{});</code>
5	<code>}</code>

En este ejemplo, la condición "edad >= 18" se evalúa. Como 20 es mayor o igual a 18, la condición es verdadera y se ejecuta el código dentro de las llaves, mostrando el mensaje "Eres mayor de edad". Si la edad hubiera sido 15, la condición sería falsa y el mensaje nunca se mostraría.

Pero ¿qué pasa si queremos hacer algo diferente cuando la condición es falsa? Para eso usamos la cláusula else (si no). El else proporciona un camino alternativo que se ejecuta cuando la condición del if es falsa:

1	const edad: u8 = 15;
2	
3	if (edad ≥ 18) {
4	std.debug.print("Eres mayor de edad\n", .{});
5	} else {
6	std.debug.print("Eres menor de edad\n", .{});
7	}

Ahora el programa tiene dos caminos posibles: si la edad es 18 o más, muestra un mensaje; si no, muestra otro mensaje diferente. Exactamente uno de los dos bloques se ejecutará, nunca ambos y nunca ninguno. Esta es una garantía importante que hace que el código sea más predecible.

1	const nota: u8 = 85;
2	
3	if (nota ≥ 90) {
4	std.debug.print("Excelente\n", .{});
5	} else if (nota ≥ 80) {
6	std.debug.print("Muy bien\n", .{});
7	} else if (nota ≥ 70) {
8	std.debug.print("Bien\n", .{});
9	} else if (nota ≥ 60) {
10	std.debug.print("Suficiente\n", .{});
11	} else {
12	std.debug.print("Insuficiente\n", .{});
13	}

Con una nota de 85, el programa primero verifica si es ≥ 90 (falso), luego si es ≥ 80 (verdadero), y ejecuta "Muy bien". Una vez que encuentra una condición verdadera, ignora todas las demás, incluso si también fueran verdaderas.

En Zig, las llaves {} son obligatorias para los bloques de if, else if, y else, incluso si solo contienen una línea de código. Esto es diferente de algunos otros lenguajes que permiten omitirlas para bloques de una sola línea. Esta regla hace el código más consistente y previene ciertos tipos de errores comunes.

Forma	Sintaxis	Uso
If simple	if (cond) { }	Un solo camino condicional
if-else	if (cond) { } else { }	Dos caminos alternativos
if-else if-else	if () { } else if () { } else { }	Múltiples condiciones
if anidado	if () { if () { } }	Condiciones dependientes

Tabla 16 - Variantes de la Sentencia if
(Fuente: Propia)

Puedes combinar múltiples condiciones usando los operadores lógicos que aprendiste en el capítulo anterior. Esto te permite crear condiciones complejas que verifican múltiples requisitos a la vez:

1	<code>const edad: u8 = 25;</code>
2	<code>const tiene_licencia: bool = true;</code>
3	
4	<code>if (edad ≥ 18 and tiene_licencia) {</code>
5	<code>std.debug.print("Puede conducir\n", .{});</code>
6	<code>} else {</code>
7	<code>std.debug.print("No puede conducir\n", .{});</code>
8	<code>}</code>

También puedes anidar sentencias if dentro de otras sentencias if. Esto es útil cuando necesitas verificar condiciones adicionales solo si una condición principal ya se cumplió. Sin embargo, demasiado anidamiento puede hacer el código difícil de leer, así que úsalo con moderación:

1	<code>const usuario_registrado: bool = true;</code>
2	<code>const es_premium: bool = false;</code>
3	
4	<code>if (usuario_registrado) {</code>
5	<code>std.debug.print("Bienvenido de vuelta\n", .{});</code>
6	<code>if (es_premium) {</code>
7	<code>std.debug.print("Tienes acceso completo\n", .{});</code>
8	<code>} else {</code>
9	<code>std.debug.print("Actualiza para más funciones\n", .{});</code>
10	<code>}</code>
11	<code>} else {</code>
12	<code>std.debug.print("Por favor regístrate\n", .{});</code>
13	<code>}</code>

Cuando necesitas comparar una variable contra muchos valores posibles, usar múltiples "else if" puede volverse tedioso y difícil de leer. Para estos casos, Zig ofrece la sentencia

switch, que es más concisa y clara cuando tienes muchas opciones. El switch evalúa una expresión y ejecuta el código correspondiente al valor que coincida:

1	const dia: u8 = 3;
2	
3	const nombre_dia = switch (dia) {
4	1 ⇒ "Lunes",
5	2 ⇒ "Martes",
6	3 ⇒ "Miércoles",
7	4 ⇒ "Jueves",
8	5 ⇒ "Viernes",
9	6 ⇒ "Sábado",
10	7 ⇒ "Domingo",
11	else ⇒ "Día inválido",
12	};

En este ejemplo, el switch evalúa la variable "día" y compara su valor contra cada caso listado. Como día es 3, el resultado es "Miércoles". La flecha "=>" separa el valor del caso del resultado o código a ejecutar. La cláusula "else" al final es obligatoria en muchos casos y maneja cualquier valor que no coincida con los casos explícitos.

Una característica poderosa del switch de Zig es que puede usarse como expresión, lo que significa que produce un valor que puedes asignar a una variable. Esto es exactamente lo que hicimos en el ejemplo anterior. Pero también puedes usar switch como sentencia para ejecutar bloques de código más complejos:

1	const opcion: u8 = 2;
2	
3	switch (opcion) {
4	1 ⇒ {
5	std.debug.print("Opcion 1 seleccionada\n", .{});
6	// más código...
7	},

8	2 ⇒ {
9	std.debug.print("Opcion 2 seleccionada\n", .{});
10	// más código...
11	},
12	else ⇒ {
13	std.debug.print("Opcion no reconocida\n", .{});
14	},
15	}

También puedes agrupar múltiples valores en un solo caso si deben ejecutar el mismo código. Esto se hace separando los valores con comas:

1	const mes: u8 = 7;
2	
3	const estacion = switch (mes) {
4	12, 1, 2 ⇒ "Verano",
5	3, 4, 5 ⇒ "Otono",
6	6, 7, 8 ⇒ "Invierno",
7	9, 10, 11 ⇒ "Primavera",
8	else ⇒ "Mes invalido",
9	};

Zig también permite usar rangos en los casos del switch, lo cual es muy útil cuando trabajas con rangos de números:

1	const nota: u8 = 75;
2	
3	const calificacion = switch (nota) {
4	90...100 ⇒ "A",
5	80...89 ⇒ "B",

6	<code>70...79 ⇒ "C",</code>
7	<code>60...69 ⇒ "D",</code>
8	<code>0...59 ⇒ "F",</code>
9	<code>else ⇒ "Nota invalida",</code>
10	<code>};</code>



Consejo

Usa switch cuando tienes muchos valores específicos que comparar. Usa if/else if cuando las condiciones son más complejas y no se reducen a simples comparaciones de igualdad. El switch es generalmente más eficiente y fácil de leer para comparaciones simples.

Los bucles son estructuras que nos permiten repetir un bloque de código múltiples veces. El bucle while (mientras) es el más simple: repite un bloque de código mientras una condición siga siendo verdadera. Cuando la condición se vuelve falsa, el bucle termina y el programa continuará con el código que viene después.

1	<code>var contador: u8 = 1;</code>
2	
3	<code>while (contador ≤ 5) {</code>
4	<code>std.debug.print("Contador: {}\n", .{contador});</code>
5	<code>contador += 1;</code>
6	<code>}</code>

Este bucle imprime los números del 1 al 5. Veamos cómo funciona paso a paso: Inicialmente, contador es 1. La condición "contador <= 5" es verdadera (1 <= 5), así que entramos al bucle, imprimimos "Contador: 1" y aumentamos contador a 2. Volvemos a verificar la condición, sigue siendo verdadera (2 <= 5), así que imprimimos "Contador: 2" y aumentamos a 3. Este proceso continuó hasta que contador llega a 6, momento en que la condición "6 <= 5" es falsa y el bucle termina.

Es crucial asegurarse de que la condición eventualmente se vuelva falsa, de lo contrario tendrás un "bucle infinito" que nunca termina y congela tu programa. En el ejemplo anterior, incrementar el contador garantiza que eventualmente superará 5 y el bucle terminará.

NOTA

Si accidentalmente creas un bucle infinito y tu programa se congela, puedes detenerlo presionando Ctrl+C en la terminal. Esto enviara una señal de interrupción al programa y lo terminara.

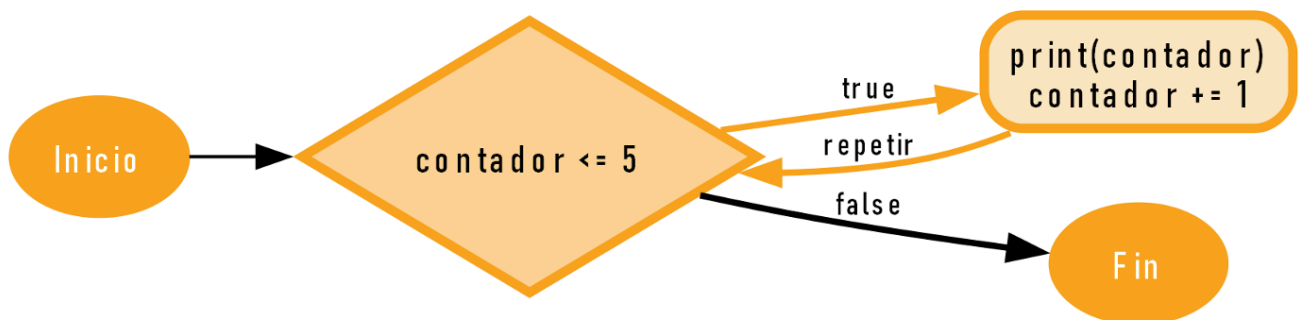


Figura 5 - Flujo de bucle while

Zig tiene una característica adicional para `while`: puedes incluir una cláusula de continuación que se ejecuta al final de cada iteración. Esto es útil para actualizar variables de control de forma más limpia:

1	<code>var i: u8 = 0;</code>
2	
3	<code>while (i < 10) : (i += 1) {</code>
4	<code> if (i % 2 == 0) {</code>
5	<code> continue; // Salta los números pares</code>
6	<code> }</code>
7	<code> std.debug.print("{} ", .{i}); // Solo imprime impares</code>
8	<code>}</code>

La expresión después de los dos puntos (`i += 1`) se ejecuta al final de cada iteración, antes de volver a verificar la condición. Esto hace el código más compacto y pone toda la lógica de control del bucle en un solo lugar.

También puedes usar la palabra clave `"break"` para salir de un bucle prematuramente, incluso si la condición aún es verdadera.

Esto es útil cuando encuentras lo que buscabas y no necesitas seguir iterando:

1	<code>var i: u8 = 0;</code>
2	<code>var encontrado: bool = false;</code>
3	
4	<code>while (i < 100) : (i += 1) {</code>
5	<code> if (i == 42) {</code>
6	<code> encontrado = true;</code>
7	<code> break; // Sale del bucle inmediatamente</code>
8	<code> }</code>
9	<code>}</code>
10	

11	if (encontrado) {
12	std.debug.print(";Encontrado en posicion {}!\n", .{i});
13	}

La palabra clave "continue" hace lo opuesto: salta el resto de la iteración actual y pasa directamente a la siguiente. Es útil cuando quieres omitir ciertos elementos, pero no terminar el bucle por completo:

1	var i: u8 = 0;
2	
3	while (i < 5) : (i += 1) {
4	std.debug.print("i = {}\n", .{i});
5	}
6	std.debug.print("{} ", .{i}); // Solo imprime impares
7	}

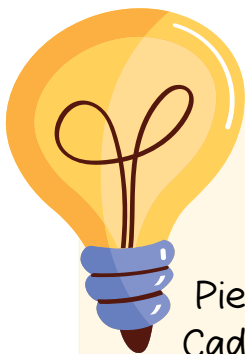
Palabra clave	Efecto	Cuando usar
Break	Termina el bucle completamente	Cuando encuentras lo que buscas
continue	Salta a la siguiente iteración	Para omitir ciertos elementos

Tabla 17 - Control de Bucles: break vs continue
(Fuente: Propia)

El bucle for en Zig está diseñado específicamente para iterar sobre colecciones de datos como arrays y slices. A diferencia del for tradicional de otros lenguajes que es básicamente un while disfrazado, el for de Zig es una herramienta más especializada y segura para recorrer elementos.

1	<code>const números = [_]u8{ 10, 20, 30, 40, 50 };</code>
2	
3	<code>for (números) número {</code>
4	<code>std.debug.print("{} ", .{número});</code>
5	<code>}</code>

Este bucle recorre cada elemento del array "números" y lo hace disponible dentro del bloque como la variable "número". La sintaxis `|número|` se llama "captura" y define el nombre de la variable que contendrá cada elemento. El bucle se ejecuta una vez por cada elemento, en orden.



Consejo

Piensa en las barras verticales `|...|` como un pequeño túnel o puerta. Cada elemento del array pasa por ese túnel uno por uno para entrar al bucle. Al pasar, le ponemos una etiqueta (el nombre de la variable) para poder usarlo dentro.

A veces también necesitas saber la posición (índice) de cada elemento. Puedes capturar tanto el elemento como su índice:

1	<code>const frutas = [_][]const u8{ "manzana", "banana", "naranja" };</code>
2	
3	<code>for (frutas, 0..) fruta, indice {</code>
4	<code>std.debug.print("{}: {s}\n", .{indice, fruta});</code>
5	<code>}</code>

El "0.." genera una secuencia de índices comenzando desde 0. El resultado de este bucle sería: "0: manzana", "1: banana", "2: naranja". Esta forma de obtener índices es muy útil cuando necesitas saber la posición de cada elemento además de su valor.

Las palabras clave break y continue también funcionan dentro de bucles for, con el mismo comportamiento que en while: break termina el bucle completamente, y continue salta a la siguiente iteración.

Si solo necesitas iterar un número específico de veces sin un array, puedes usar un rango:

1	// Imprimir números del 1 al 5
2	for (1..6) n {
3	std.debug.print("{} ", .{n});
4	}

El rango "1..6" genera los números 1, 2, 3, 4, 5 (el límite superior no se incluye). Esto es muy práctico para repetir una acción un número determinado de veces.



NOTA

¡Ojo con los puntos! En Zig, '..' (dos puntos) crea un rango exclusivo que NO incluye el número final (1..6 va del 1 al 5). En cambio, '...' (tres puntos) que vimos en el switch crea un rango inclusivo que SÍ incluye el final. Es fácil confundirlos al principio, así que mantén los ojos abiertos.

Bucle	Mejor para	Ejemplo de Uso
while	Repetir hasta condición	Esperar entrada, buscar valor
For	Iterar colecciones/rangos	Recorrer arrays, contar de 1 a N

Tabla 18 - Comparación: while vs for
(Fuente: Propia)

Puedes colocar bucles dentro de otros bucles, creando lo que se llama "bucles anidados". Esto es común cuando trabajas con estructuras bidimensionales como tablas o matrices. El bucle exterior controla las filas, y el bucle interior controla las columnas:

1	// Imprimir una tabla de multiplicar parcial
2	for (1..4) fila {
3	for (1..4) columna {
4	const producto = fila * columna;
5	std.debug.print("{} x {} = {}\t", .{fila, columna, producto});
6	}
7	std.debug.print("\n", .{});
8	}

Este código produce una tabla de multiplicar de 3x3.



Consejo

Ten cuidado con bucles anidados profundos (más de 2 o 3 niveles). Cada nivel adicional multiplica el número de iteraciones y puede hacer que tu programa sea muy lento. Si te encuentras con muchos niveles de anidamiento, considera si hay una forma más eficiente de estructurar tu código.

Cuando tienes bucles anidados, a veces quieres que break o continue afecten al bucle exterior en lugar del interior. Zig permite esto usando etiquetas de bucle:

1	const matriz = [3][3]u8{
2	.{ 1, 2, 3 },
3	.{ 4, 5, 6 },
4	.{ 7, 8, 9 },
5	};
6	
7	const objetivo: u8 = 5;
8	var encontrado = false;
9	
10	busqueda: for (matriz) fila {
11	for (fila) elemento {
12	if (elemento == objetivo) {
13	encontrado = true;
14	break :busqueda; // Sale de ambos bucles
15	}
16	}
17	}

La etiqueta "busqueda:" marca el bucle exterior. Cuando encontramos el elemento que buscamos, "break :busqueda" sale directamente del bucle etiquetado, sin necesidad de agregar lógica adicional para propagar la salida a través de los bucles anidados.

Cada vez que usas llaves {} para definir un bloque de código (ya sea en un if, while, for, o incluso por sí solo), estás creando un nuevo "ambito" o "scope".

1	<code>const exterior: i32 = 10;</code>
2	
3	<code>if (true) {</code>
4	<code> const interior: i32 = 20;</code>
5	<code> std.debug.print("Exterior: {}\n", .{exterior}); // OK</code>
6	<code> std.debug.print("Interior: {}\n", .{interior}); // OK</code>
7	<code>}</code>
8	
9	<code>std.debug.print("Exterior: {}\n", .{exterior}); // OK</code>
10	<code>// std.debug.print("Interior: {}\n", .{interior}); // ERROR: interior no existe aquí</code>

Esta regla de ámbito es muy útil porque evita que las variables "escapen" y causen confusión. Puedes reutilizar nombres de variables en diferentes bloques sin preocuparte de que interfieran entre sí. También significa que el compilador puede liberar la memoria de las variables tan pronto como el bloque termina.

Veamos un programa más sustancial que combina varias estructuras de control. Este programa simula un menú simple de calculadora que permite al usuario imaginario realizar operaciones:

1	<code>const std = @import("std");</code>
2	
3	<code>pub fn main() void {</code>
4	<code> const operaciones = [_]u8{ '+', '-', '*', '/' };</code>
5	<code> const a: f64 = 10.0;</code>
6	<code> const b: f64 = 3.0;</code>
7	
8	<code> std.debug.print("Calculadora - Demostrando estructuras de control\n", . {});</code>
9	<code> std.debug.print("=====\n", .{});</code>
10	
11	<code> for (operaciones) op {</code>

12	const resultado: f64 = switch (op) {
13	'+' ⇒ a + b,
14	'-' ⇒ a - b,
15	'*' ⇒ a * b,
16	'/' ⇒ if (b ≠ 0) a / b else 0,
17	else ⇒ 0,
18	};
19	
20	std.debug.print("{d:.1} {c} {d:.1} = {d:.2}\n", .{a, op, b, resultado});
21	}
22	
23	// Contar números pares del 1 al 20
24	var pares: u8 = 0;
25	var i: u8 = 1;
26	while (i ≤ 20) : (i += 1) {
27	if (i % 2 == 0) {
28	pares += 1;
29	}
30	}
31	std.debug.print("\nNúmeros pares del 1 al 20: {}\n", .{pares});
32	}

En este capítulo hemos aprendido las estructuras de control qué le dan a tus programas la capacidad de tomar decisiones y repetir acciones. Con la sentencia if, tu programa puede elegir qué código ejecutar basándose en condiciones. Puedes encadenar múltiples condiciones con else if, y proporcionar un camino por defecto con else.

La sentencia switch es perfecta cuando necesitas comparar una variable contra muchos valores posibles, ofreciendo una sintaxis más limpia que múltiples else if. Puede usarse como expresión para producir valores directamente.

Los bucles `while` y `for` te permiten repetir código. El `while` es más flexible y continúa mientras una condición sea verdadera. El `for` es más especializado para iterar sobre colecciones y rangos. Las palabras clave `break` y `continue` te dan control adicional sobre cómo se ejecutan los bucles, y las etiquetas te permiten afectar bucles exteriores desde bucles anidados.

Finalmente, aprendiste sobre el ámbito de las variables y cómo los bloques de código crean zonas donde las variables existen y luego desaparecen.

EJERCICIOS

Pon a prueba tus conocimientos con los siguientes ejercicios:

CLASIFICADOR DE EDADES

Escribe un programa que clasifique a una persona según su edad: 'Bebé' (0-2 años), 'Niño' (3-12), 'Adolescente' (13-17), 'Adulto joven' (18-30), 'Adulto' (31-60), 'Adulto mayor' (61+). Usa una estructura if/else if/else.

Ejercicio

NOMBRE DEL MES

Escribe un programa que use switch para convertir un número de mes (1-12) a su nombre. Para los meses 1, 2, 3 también indica que son del primer trimestre.

Ejercicio

SUMA CON WHILE

Escribe un programa que use un bucle while para calcular la suma de todos los números del 1 al 100. Muestra el resultado final.

Ejercicio

3

CAPÍTULO 6

Funciones

A medida que tus programas crecen, te encontrarás escribiendo el mismo código una y otra vez. Quizás necesitas calcular el área de un círculo en varios lugares de tu programa, o validar que un dato ingresado sea correcto en múltiples puntos. Copiar y pegar el mismo código no solo es tedioso, sino que crea un problema de mantenimiento: si descubres un error o quieres hacer una mejora, tendrás que modificar cada copia individualmente. Las funciones resuelven este problema permitiéndote escribir un bloque de código una sola vez y reutilizarlo cuantas veces necesites.

Una función es como una mini-máquina dentro de tu programa: recibe ciertos datos de entrada (llamados parámetros), realiza algún procesamiento con ellos, y opcionalmente produce un resultado (llamado valor de retorno). Piensa en una licuadora: le pones frutas (entrada), hace su trabajo (procesamiento), y te entrega un batido (salida). No necesitas saber exactamente cómo funcionan los motores y cuchillas internas; solo necesitas saber qué poner y qué esperar como resultado. Las funciones funcionan de manera similar: encapsulan la complejidad y te permiten usarlas sin preocuparte por los detalles internos.

Ya has estado usando funciones sin quizás darte cuenta. La función `main()` que aparece en todos nuestros programas es precisamente eso: una función. También has llamado a `std.debug.print`, que es otra función proporcionada por la librería estándar de Zig. En este capítulo aprenderás a crear tus propias funciones, lo cual te permitirá organizar tu código de manera más clara, evitar repetición, y construir programas más grandes y mantenibles.

En Zig, las funciones se definen usando la palabra clave `fn` seguida del nombre de la función, sus parámetros entre paréntesis, el tipo de retorno, y finalmente el cuerpo de la función entre llaves. La sintaxis básica es:

1	<code>fn nombre_función(parámetro1: tipo1, parámetro2: tipo2) tipo_retorno {</code>
2	<code> // cuerpo de la función</code>
3	<code> return valor;</code>
4	<code>}</code>

Componente	Ejemplo	Descripción
Palabra clave	fn	Indica definición de función
Nombre	sumar	Identificador de la función
Parámetros	(a: i32, b: i32)	Datos de entrada con tipos
Tipo retorno	i32	Tipo del valor devuelto
Cuerpo	{ return a + b; }	Código a ejecutar

Tabla 19 - Componentes de una Función
(Fuente: Propia)



Figura 6 - Máquina de funciones

Empecemos con un ejemplo simple: una función que suma dos números enteros y devuelve el resultado:

1	fn sumar(a: i32, b: i32) i32 {
2	return a + b;
3	}

Esta función se llama "sumar" y tiene dos parámetros: "a" y "b", ambos de tipo i32. El tipo después de los paréntesis (i32) indica que la función devuelve un entero de 32 bits. La palabra clave "return" especifica el valor que la función devuelve a quien la llamó. Para usar esta función, simplemente la llamamos por su nombre y le pasamos los valores necesarios:

1	const resultado = sumar(5, 3); // resultado es 8
2	const otro = sumar(100, 200); // otro es 300

Cuando llamas a una función, Zig ejecuta el código dentro de ella con los valores que proporcionaste. Los valores 5 y 3 se asignan a los parámetros "a" y "b" respectivamente, se calcula a + b (que es 8), y ese valor se devuelve. El valor devuelto se asigna a la variable

"resultado". Este proceso de pasar valores y recibir un resultado ocurre cada vez que llamas a la función.

Las funciones pueden tener cualquier cantidad de parámetros, desde ninguno hasta muchos. Una función sin parámetros tendría paréntesis vacíos:

1	fn obtener_pi() f64 {
2	return 3.14159265358979;
3	}
4	
5	const pi = obtener_pi(); // pi es 3.14159...

No todas las funciones necesitan devolver un valor. Algunas funciones realizan acciones (como imprimir en pantalla o modificar datos) sin producir un resultado que necesites capturar. Para indicar que una función no devuelve nada, usamos el tipo "void" como tipo de retorno:

1	fn saludar(nombre: []const u8) void {
2	std.debug.print("¡Hola, {s}!\n", .{nombre});
3	}
4	
5	saludar("Maria"); // Imprime: ¡Hola, Maria!

Tipo	Uso	Ejemplo
void	Sin valor de retorno	fn saludar() void
i32, f64, etc.	Retorna valor específico	fn sumar() i32
!T (error unión)	Puede retornar error	fn dividir() !f64
?T (opcional)	Puede retornar null	fn buscar() ?i32

Tabla 20 - Tipos de Retorno de Funciones
(Fuente: Propia)

Las funciones void no usan la sentencia return para devolver un valor (aunque pueden usar "return;" sin valor para terminar la ejecución prematuramente). Simplemente realizan su trabajo y cuando llegan al final, la ejecución vuelve automáticamente al punto donde fueron llamadas.



NOTA

La función main() que define el punto de entrada de tu programa típicamente tiene tipo de retorno void ("pub fn main() void"), aunque también puede retornar un código de error en programas más avanzados. Por ahora, usaremos la version void.

Los parámetros son la forma en que pasas información a una función. En Zig, todos los parámetros se pasan "por valor" por defecto, lo que significa que la función recibe una copia del valor. Modificar el parámetro dentro de la función no afecta al valor original:

1	fn intentar_modificar(n: i32) void {
2	// Esta es una copia local, el original no cambia
3	var local = n;
4	local = 999; // Solo afecta a la copia local
5	std.debug.print("Dentro: {}\n", .{local});
6	}
7	
8	var número: i32 = 42;
9	intentar_modificar(número);
10	std.debug.print("Fuera: {}\n", .{número}); // Sigue siendo 42

Los parámetros en Zig son inmutables por defecto dentro de la función. Esto significa que no puedes reasignar un parámetro directamente. Si necesitas modificar el valor dentro de la función, debes primero copiarlo a una variable local:

1	<code>fn procesar(valor: i32) i32 {</code>
2	<code> // valor = valor + 1; // ERROR: no se puede modificar el parámetro</code>
3	<code> var temporal = valor; // Crear una copia mutable</code>
4	<code> temporal += 1;</code>
5	<code> return temporal;</code>
6	<code>}</code>

Para funciones simples que solo devuelven el resultado de una expresión, puedes omitir las llaves y usar una sintaxis más concisa con la flecha "=>":

1	<code>fn cuadrado(n: i32) i32 {</code>
2	<code> return n * n;</code>
3	<code>}</code>
4	
5	<code>// También se puede escribir más concisamente en algunos casos:</code>
6	<code>fn doble(n: i32) i32 {</code>
7	<code> const resultado = n * 2;</code>
8	<code> return resultado;</code>
9	<code>}</code>

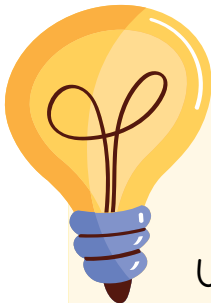
Zig también permite usar bloques como expresiones, lo cual es útil dentro de funciones cuando necesitas hacer cálculos intermedios antes de retornar:

1	<code>fn calcular_promedio(a: f64, b: f64, c: f64) f64 {</code>
2	<code> const suma = a + b + c;</code>
3	<code> const promedio = suma / 3.0;</code>
4	<code> return promedio;</code>
5	<code>}</code>

Las funciones pueden llamar a otras funciones, lo cual te permite construir programas complejos a partir de piezas simples. Esta es una de las ideas más poderosas en programación: descomponer un problema grande en problemas más pequeños, resolver cada uno con una función, y luego combinarlas.

1	<code>fn cuadrado(n: i32) i32 {</code>
2	<code> return n * n;</code>
3	<code>}</code>
4	
5	<code>fn suma_de_cuadrados(a: i32, b: i32) i32 {</code>
6	<code> return cuadrado(a) + cuadrado(b);</code>
7	<code>}</code>
8	
9	<code>const resultado = suma_de_cuadrados(3, 4); // 9 + 16 = 25</code>

En este ejemplo, `suma_de_cuadrados` no implementa la lógica del cuadrado directamente; delega ese trabajo a la función `cuadrado`. Esto hace que el código sea más modular y fácil de entender. Si algún día necesitas cambiar cómo se calcula el cuadrado (quizás para agregar validación o usar un algoritmo diferente), solo necesitas modificar la función `cuadrado`, y todas las funciones que la usan se beneficiarán automáticamente.



Consejo

Una buena práctica es que cada función haga una sola cosa bien. Si una función está haciendo demasiadas cosas, considera dividirla en varias funciones más pequeñas. Esto hace el código más fácil de probar, entender y mantener.

Una función puede incluso llamarse a sí misma, un concepto llamado recursividad. Esto es útil para problemas que naturalmente se definen en términos de versiones más pequeñas de sí mismos. El ejemplo clásico es el cálculo del factorial: el factorial de n (escrito $n!$) es n multiplicado por el factorial de $n-1$.

1	<code>fn factorial(n: u64) u64 {</code>
2	<code> if (n ≤ 1)</code>
3	<code> return 1; // Caso base: 0! = 1! = 1</code>
4	<code> }</code>
5	<code> return n * factorial(n - 1); // Caso recursivo</code>
6	<code>}</code>
7	
8	<code>const resultado = factorial(5); // 5 * 4 * 3 * 2 * 1 = 120</code>

Veamos cómo funciona paso a paso cuando llamamos `factorial(5)`: La función verifica si $5 \leq 1$ (falso), así que retorna $5 * \text{factorial}(4)$. Para calcular `factorial(4)`, retorna $4 * \text{factorial}(3)$. Esto continúa hasta `factorial(1)`, que retorna 1 porque $1 \leq 1$ es verdadero. Luego los resultados se multiplican hacia atrás: $2*1=2$, $3*2=6$, $4*6=24$, $5*24=120$.

Toda función recursiva debe tener un "caso base" que no haga una llamada recursiva, de lo contrario la función se llamaría a sí misma infinitamente hasta agotar la memoria (causando un "stack overflow", lo cual es una situación en que la pila de llamadas se llena porque la recursión no tiene fin, agotando la memoria disponible). En el ejemplo de la factorial, el caso base es cuando $n \leq 1$.



NOTA

La recursividad es elegante pero puede ser menos eficiente que los bucles para algunos problemas. Cada llamada recursiva consume memoria, y las funciones muy profundamente recursivas pueden causar problemas. Usa la recursividad cuando haga el código más claro, pero considera alternativas iterativas para casos donde el rendimiento es crítico.

En Zig, la visibilidad de las funciones se controla con la palabra clave "pub" (público). Una función marcada como "pub" puede ser llamada desde otros archivos que importen el módulo actual. Una función sin "pub" es privada al archivo donde se define.

1	// Esta función es publica, puede usarse desde otros archivos
2	pub fn función_publica() void {
3	función_privada(); // Puede llamar funciones privadas del mismo archivo
4	}
5	
6	// Esta función es privada, solo visible en este archivo
7	fn función_privada() void {
8	std.debug.print("Soy privada\n", .{});
9	}

Modificador	Sintaxis	Accesible desde
Público	pub fn nombre()	Cualquier archivo que importe
Privado	fn nombre()	Solo el archivo actual

Tabla 21 - Visibilidad de Funciones
(Fuente: Propia)

La función main() siempre se marca como "pub" porque es el punto de entrada del programa y debe ser visible externamente. Para programas simples de un solo archivo, la distinción entre público y privado no importa mucho, pero se vuelve importante cuando organizas tu código en múltiples archivos o creas bibliotecas.

Aspecto	pub fn (Publica)	fn (Privada)
Accesible desde	Otros archivos/módulos	Solo este archivo
Exportada	Sí	No
Uso	API pública	Implementación interna

Tabla 22 - Visibilidad de Funciones
(Fuente: Propia)

Zig proporciona funciones integradas (builtins) que comienzan con el símbolo @. Estas funciones están implementadas directamente en el compilador y proporcionan operaciones fundamentales que no podrían expresarse eficientemente como funciones normales.

1	const minimo = @min(5, 3); // 3
2	const maximo = @max(5, 3); // 5
3	const absoluto = @abs(-42); // 42
4	const tipo = @TypeOf(42); // El tipo de la expresión
5	const tamaño = @sizeof(i32); // 4 (bytes que ocupa un i32)

Función	Descripción	Ejemplo
@min(a, b)	Retorna el menor	@min(5, 3) → 3
@max(a, b)	Retorna el mayor	@max(5, 3) → 5
@abs(x)	Valor absoluto	@abs(-42) → 42
@TypeOf(x)	Tipo de la expresión	@TypeOf(42) → i32
@sizeof(T)	Tamaño en bytes	@sizeof(i32) → 4

Tabla 23 - Funciones Integradas Comunes
(Fuente: Propia)

Estas funciones integradas son muy útiles para operaciones matemáticas comunes, información sobre tipos, y otras operaciones de bajo nivel. A medida que avances, descubrirás más funciones @ que te ayudarán en diferentes situaciones.

En programas reales, las funciones a veces no pueden completar su trabajo: un archivo podría no existir, una conexión de red podría fallar, o los datos de entrada podrían ser inválidos. Zig tiene un sistema de manejo de errores integrado que veremos en detalle en un capítulo posterior, pero aquí hay una introducción básica.

Una función que puede fallar indica esto en su tipo de retorno usando "!" seguido del tipo normal de retorno. El tipo "!"T" (llamado "error unión") significa "puede devolver un valor de tipo T, o puede devolver un error":

1	<code>fn dividir(a: f64, b: f64) !f64 {</code>
2	<code> if (b == 0) {</code>
3	<code> return error.DivisiónPorCero;</code>
4	<code> }</code>
5	<code> return a / b;</code>
6	<code>}</code>

Cuando llamas a una función que puede fallar, debes decidir cómo manejar el posible error. Una opción es usar "try", que propaga el error al llamador si ocurre, o "catch" para manejar el error en el lugar:

1	<code>// Usando try (propaga el error si ocurre)</code>
2	<code>const resultado = try dividir(10, 2);</code>
3	
4	<code>// Usando catch (maneja el error localmente)</code>
5	<code>const resultado2 = dividir(10, 0) catch err {</code>
6	<code> std.debug.print("Error: {}\n", .{err});</code>
7	<code> return; // o proporcionar un valor por defecto</code>
8	<code>};</code>



Consejo

Piensa en 'try' como el juego de la 'patata caliente': si recibes un error (la patata), no te lo quedas, se lo pasas inmediatamente a quien te llamó. En cambio, 'catch' es atrapar la patata, enfriarla y manejar la situación tú mismo.

El manejo de errores es un tema importante que cubriremos completamente en el capítulo dedicado a ello. Por ahora, solo debes saber que Zig tiene soporte de primera clase para funciones que pueden fallar.

A diferencia de algunos lenguajes, en Zig no importa en qué orden definas tus funciones. Puedes llamar a una función que está definida más abajo en el archivo, porque Zig analiza todo el archivo antes de compilar. Esto te permite organizar tu código de la manera que tenga más sentido, sin preocuparte por el orden:

1	<code>fn main() void {</code>
2	<code> const x = segunda(); // Llama a una función definida después</code>
3	<code> std.debug.print("Resultado: {}\n", .{x});</code>
4	<code>}</code>
5	
6	<code>fn segunda() i32 {</code>
7	<code> return primera() + 1;</code>
8	<code>}</code>
9	
10	<code>fn primera() i32 {</code>
11	<code> return 41;</code>
12	<code>}</code>

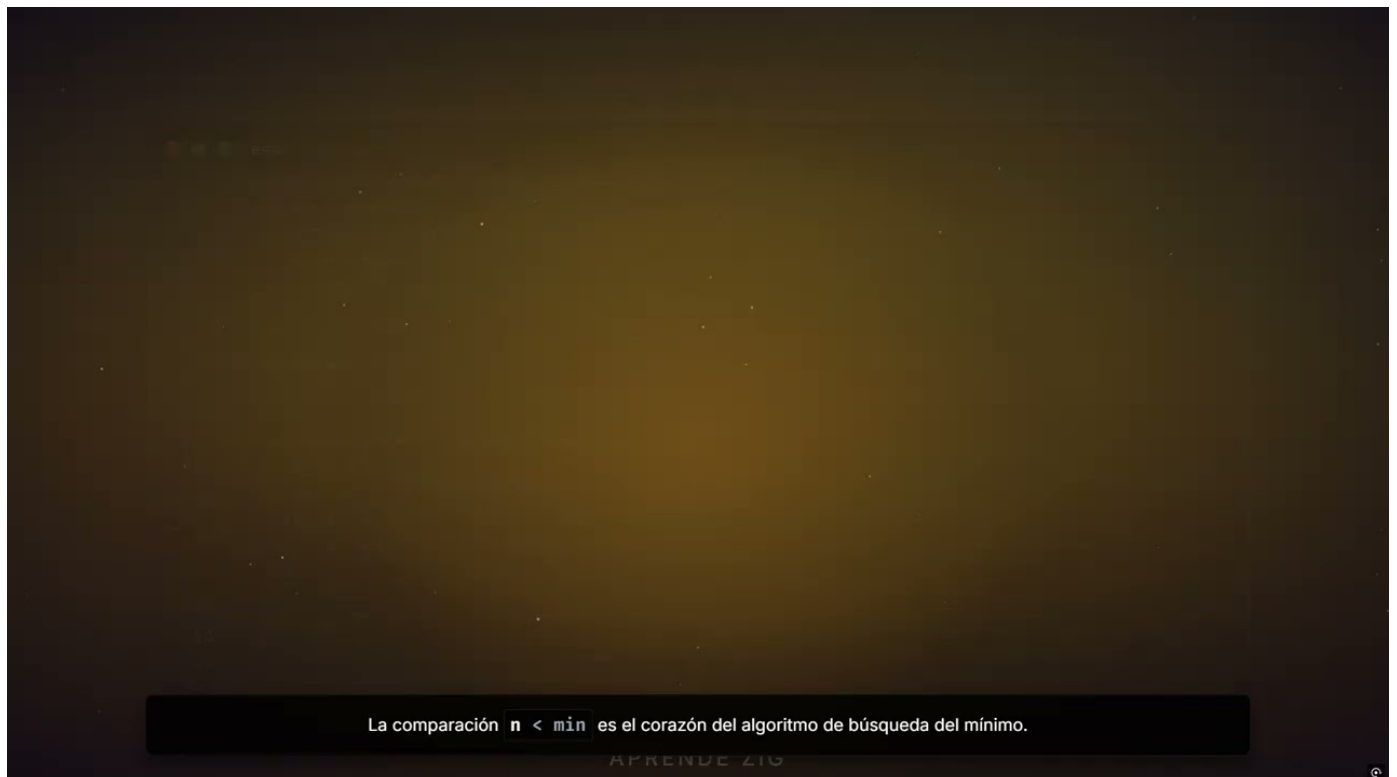
Esto es muy conveniente porque te permite estructurar tu código con las funciones más importantes (como main) al principio del archivo, seguidas de las funciones de apoyo, sin necesidad de declaraciones adelantadas o un orden específico.

Veamos un programa completo que utiliza funciones para resolver un problema práctico: calcular estadísticas básicas de un conjunto de números. Este ejemplo muestra cómo las funciones trabajan juntas para crear un programa útil:

1	<code>const std = @import("std");</code>
2	
3	<code>fn sumar_array(números: []const i32) i32 {</code>
4	<code> var suma: i32 = 0;</code>
5	<code> for (números) n {</code>
6	<code> suma += n;</code>
7	<code> }</code>
8	<code> return suma;</code>
9	<code>}</code>
10	
11	<code>fn encontrar_maximo(números: []const i32) i32 {</code>
12	<code> var max = números[0];</code>
13	<code> for (números[1..]) n {</code>
14	<code> if (n > max) {</code>
15	<code> max = n;</code>
16	<code> }</code>
17	<code> }</code>
18	<code> return max;</code>
19	<code>}</code>
20	
21	<code>const std = @import("std");</code>
22	
23	<code>fn sumar_array(números: []const i32) i32 {</code>
24	<code> var suma: i32 = 0;</code>
25	<code> for (números) n {</code>
26	<code> suma += n;</code>
26	<code> }</code>
27	<code>}</code>
28	<code> return min;</code>
29	<code>}</code>

30	
31	fn calcular_promedio(números: []const i32) f64 {
32	const suma = sumar_array(números);
33	const cantidad: f64 = @floatFromInt(números.len);
34	return @as(f64, @floatFromInt(suma)) / cantidad;
35	}
36	
37	pub fn main() void {
38	const datos = [_]i32{ 45, 23, 67, 12, 89, 34, 56 };
39	
40	std.debug.print("Estadísticas del conjunto de datos:\n", .{});
41	std.debug.print("Suma: {}\n", .{sumar_array(&datos)});
42	std.debug.print("Maximo: {}\n", .{encontrar_maximo(&datos)});
43	std.debug.print("Minimo: {}\n", .{encontrar_minimo(&datos)});
44	std.debug.print("Promedio: {d:.2}\n", .{calcular_promedio(&datos)});
45	}

Video 6 - Funciones en Zig



Este programa define cuatro funciones especializadas, cada una responsable de un cálculo específico. La función `main()` las orquesta para producir un informe estadístico. Observa cómo `calcular_promedio` reutiliza `sumar_array` en lugar de reimplementar la suma. Este es un buen ejemplo de cómo las funciones promueven la reutilización de código.

En este capítulo aprendiste sobre las funciones, una de las herramientas más importantes para organizar y reutilizar código. Viste cómo definir funciones con la palabra clave `fn`, cómo especificar parámetros y tipos de retorno, y cómo llamar a funciones pasando argumentos.

Exploramos funciones que retornan valores y funciones `void` que solo realizan acciones. Aprendiste que los parámetros se pasan por valor y son inmutables dentro de la función, lo que hace el código más predecible. También vimos cómo las funciones pueden llamarse entre sí, incluyendo la recursividad donde una función se llama a sí misma.

Cubrimos la visibilidad de funciones (`pub` vs `privada`), las funciones integradas que empiezan con `@`, y tuvimos un primer vistazo al manejo de errores en funciones. Finalmente, vimos que el orden de definición no importa en Zig, lo que te da flexibilidad para organizar tu código.

En el próximo capítulo aprenderemos sobre arrays y slices, las estructuras de datos que te permiten almacenar y manipular colecciones de valores. Con arrays y las funciones que ya conoces, podrás escribir programas que procesen grandes cantidades de datos de manera eficiente.

EJERCICIOS

Pon a prueba tus conocimientos con los siguientes ejercicios:

FUNCIÓN DE ÁREA

Escribe una función llamada 'area_rectángulo' que reciba el ancho y alto como parámetros f64 y retorne el area. Luego escribe otra función 'area_cuadrado' que reciba solo el lado y use 'area_rectángulo' internamente para calcular el área.

Ejercicio

FUNCIÓN DE MÁXIMO DE TRES

Escribe una función 'maximo_tres' que reciba tres números enteros y retorne el mayor de los tres. Usa comparaciones y la estructura if/else.

Ejercicio

FUNCIÓN ES_PRIMO

Escribe una función 'es_primo' que reciba un número entero positivo y retorne true si es primo, false si no lo es. Un número es primo si solo es divisible por 1 y por sí mismo.

Ejercicio

3

FIBONACCI RECURSIVO

Escribe una función recursiva 'fibonacci' que calcule el n-ésimo número de la secuencia de Fibonacci. Recuerda que $F(0)=0$, $F(1)=1$, y $F(n)=F(n-1)+F(n-2)$ para $n>1$.

Ejercicio

4

CAPÍTULO 7

Arrays y Slices

Hasta ahora hemos trabajado con variables individuales: un número, un texto, un booleano. Pero en el mundo real, frecuentemente necesitas manejar colecciones de datos relacionados. Imagina que estás construyendo un programa para llevar las notas de una clase. ¿Crearías una variable separada para cada estudiante? ¿`nota_estudiante1`, `nota_estudiante2`, `nota_estudiante3`...? Esto sería tedioso e imposible de mantener para una clase de 30 o 100 estudiantes. Los arrays y slices resuelven este problema permitiéndote agrupar múltiples valores del mismo tipo bajo un solo nombre.

Un array es una colección ordenada de elementos del mismo tipo, almacenados en posiciones consecutivas de memoria. Puedes pensar en un array como una fila de casilleros numerados: cada casillero puede contener un elemento, y puedes acceder a cualquier casillero si conoces su número (llamado índice). El primer elemento está en el índice 0, el segundo en el índice 1, y así sucesivamente.

Un slice, por otro lado, es una vista flexible sobre una secuencia de elementos. Mientras que un array tiene un tamaño fijo conocido en tiempo de compilación, un slice puede referirse a cualquier porción de un array, y su tamaño puede variar. Los slices son increíblemente útiles para pasar colecciones a funciones sin necesidad de copiar todos los datos.

Características	Array	Slice
Tamaño	Fijo en compilación	Variable en ejecución
Tipo	[N]T (ej: [5]i32)	[]T (ej: []i32)
Memoria	Contiene los datos	Referencia a datos
Uso Típico	Datos de tamaño conocido	Parámetros de función

Tabla 24 - Arrays vs Slices
(Fuente: Propia)



Figura 7 - Visualización del Array números

En Zig, los arrays se definen especificando el número de elementos y el tipo de cada elemento. La sintaxis usa corchetes con el tamaño antes del tipo. Veamos cómo crear un array de 5 números enteros:

```
1 const números: [5]i32 = [5]i32{ 10, 20, 30, 40, 50 };
```

Aquí, `[5]i32` significa "un array de 5 elementos de tipo `i32`". Los valores se especifican entre llaves, separados por comas. Zig te obliga a proporcionar exactamente tantos elementos como el tamaño del array indica; ni más, ni menos.

Zig puede inferir el tamaño del array si usas el guion bajo en lugar del número:

```
1 const letras = [_]u8{ 'a', 'b', 'c', 'd' }; // Zig infiere [4]u8
```

El guion bajo le dice a Zig "calcula cuántos elementos hay". Esto es conveniente porque no tienes que contar manualmente los elementos y actualizar el número si agregas o quitas elementos. Sin embargo, recuerda que el tamaño sigue siendo fijo una vez compilado el programa.

```
1 const ceros: [10]i32 = [_]i32{0} ** 10; // 10 ceros
```

```
2 const unos = [_]u8{1} ** 5; // [1, 1, 1, 1, 1]
```

El operador `**` repite el patrón el número especificado de veces. Esto es muy útil para inicializar arrays grandes con un valor por defecto.

Sintaxis	Ejemplo	Resultado
Tamaño explícito	<code>[5]i32{ 1, 2, 3, 4, 5 }</code>	Array de 5 elementos
Tamaño inferido	<code>[_]i32{ 1, 2, 3 }</code>	Array de 3 elementos
Repetición	<code>[_]i32{0} ** 10</code>	10 ceros

Tabla 25 - Formas de Crear Arrays
(Fuente: Propia)

Para acceder a un elemento específico de un array, usas el índice entre corchetes. Recuerda que los índices empiezan en 0:

1	<code>const frutas = [_][]const u8{ "manzana", "banana", "naranja", "uva" };</code>
2	
3	<code>const primera = frutas[0]; // "manzana"</code>
4	<code>const tercera = frutas[2]; // "naranja"</code>
5	<code>const ultima = frutas[3]; // "uva"</code>

Zig verifica en tiempo de compilación (cuando es posible) y en tiempo de ejecución que los índices están dentro del rango válido. Si intentas acceder a `frutas[10]` en un array de 4 elementos, Zig detectará el error y te lo indicará, en lugar de permitir que tu programa acceda a memoria inválida. Esta verificación de límites es una característica de seguridad importante.

NOTA

Los índices fuera de rango son una fuente muy común de errores en programación, conocidos como 'buffer overflows'. Zig te protege de estos errores verificando automáticamente los límites del array. Esto puede tener un pequeño costo de rendimiento, pero previene errores graves y vulnerabilidades de seguridad.

También puedes acceder al último elemento usando una sintaxis especial con la propiedad `len` (longitud) del array:

1	<code>const números = [_]i32{ 10, 20, 30, 40, 50 };</code>
2	<code>const cantidad = números.len; // 5</code>
3	<code>const último = números[números.len - 1]; // 50</code>

Para modificar elementos de un array, este debe ser declarado con `"var"` en lugar de `"const"`. Luego puedes asignar nuevos valores a posiciones específicas:

1	<code>var temperaturas = [_]f32{ 20.5, 22.3, 19.8, 21.0 };</code>
2	
3	<code>temperaturas[0] = 23.5; // Cambia el primer elemento</code>
4	<code>temperaturas[2] = 18.0; // Cambia el tercer elemento</code>
5	
6	<code>// Ahora el array es: { 23.5, 22.3, 18.0, 21.0 }</code>

Es importante entender que solo puedes modificar los valores dentro del array, no su tamaño. Un array declarado con 4 elementos siempre tendrá 4 elementos. Si necesitas una colección que pueda crecer o reducirse, Zig ofrece otras estructuras de datos que veremos más adelante.

Una de las operaciones más comunes con arrays es recorrer todos sus elementos. El bucle `for` es perfecto para esto, ya que está diseñado específicamente para iterar sobre colecciones:

1	<code>const nombres = [_][]const u8{ "Ana", "Luis", "Maria", "Carlos" };</code>
2	
3	<code>for (nombres) nombre {</code>
4	<code>std.debug.print("Hola, {s}!\n", .{nombre});</code>
5	<code>}</code>

Cada iteración del bucle asigna un elemento del array a la variable "nombre" (capturada entre barras verticales `Inombre`). El bucle se ejecuta una vez por cada elemento, en orden de izquierda a derecha.

Si también necesitas el índice de cada elemento, puedes combinar el array con un rango de índices:

1	<code>const valores = [_]i32{ 100, 200, 300, 400 };</code>
2	
3	<code>for (valores, 0..) valor, indice {</code>
4	<code>std.debug.print("valores[{}] = {}\n", .{indice, valor});</code>
5	<code>}</code>

El "0.." genera índices comenzando desde 0. Zig automáticamente genera tantos índices como elementos haya en el array. Esta es una forma muy limpia y segura de obtener tanto el valor como su posición.

Los slices son vistas sobre secuencias de elementos. A diferencia de los arrays que tienen un tamaño fijo conocido en tiempo de compilación, los slices tienen un tamaño determinado en tiempo de ejecución. Esto los hace más flexibles para pasar datos entre funciones.

crear un slice a partir de un array usando la sintaxis de rango:

1	<code>const array = [_]i32{ 10, 20, 30, 40, 50, 60, 70 };</code>
2	
3	<code>const slice_completo = array[0..];</code> // Todo el array como slice
4	<code>const primeros_tres = array[0..3];</code> // Elementos 0, 1, 2
5	<code>const del_medio = array[2..5];</code> // Elementos 2, 3, 4
6	<code>const últimos_dos = array[5..];</code> // Elementos 5, 6

La sintaxis `[inicio..fin]` crea un slice que incluye los elementos desde el índice "inicio" hasta el índice "fin - 1" (el índice final no se incluye). Si omites el inicio, se asume 0. Si omites el final, se incluye hasta el último elemento.

Expresión	Descripción	Elementos incluidos
array[0..3]	Del índice 0 al 2	3 elementos
array[2..5]	Del índice 2 al 4	3 elementos
array[0..]	Desde el inicio	Todos
array[5..]	Desde índice 5	Hasta el final

Tabla 26 - Sintaxis de Slicing
(Fuente: Propia)

El tipo de un slice se escribe `[]T`, donde `T` es el tipo de los elementos. Por ejemplo, `[]i32` es un slice de enteros de 32 bits. Observa que no hay número entre los corchetes, indicando que el tamaño no es parte del tipo:

1	<code>const array: [5]i32 = .{ 1, 2, 3, 4, 5 }; // Tipo: [5]i32</code>
2	<code>const slice: []const i32 = &array; // Tipo: []const i32</code>

NOTA

El `&` antes del array obtiene un slice de todo el array. Cuando pasas un array a una función que espera un slice, necesitas usar `&` para obtener la referencia. Esto es un concepto relacionado con punteros que exploraremos en el próximo capítulo.

Los slices brillan especialmente cuando trabajas con funciones. Una función que acepta un slice puede trabajar con arrays de cualquier tamaño, haciéndola mucho más flexible:

1	<code>fn sumar_elementos(números: []const i32) i32 {</code>
2	<code>var suma: i32 = 0;</code>

3	for (números) n {
4	suma += n;
5	}
6	return suma;
7	}
8	
9	pub fn main() void {
10	const array_pequeño = [_]i32{ 1, 2, 3 };
11	const array_grande = [_]i32{ 10, 20, 30, 40, 50, 60 };
12	
13	// La misma función trabaja con arrays de diferentes tamaños
14	const suma1 = sumar_elementos(&array_pequeño); // 6
15	const suma2 = sumar_elementos(&array_grande); // 210
16	}

La función `sumar_elementos` acepta un slice `[]const i32`, lo que significa que puede recibir cualquier secuencia de enteros, sin importar cuántos elementos tenga. El "const" indica que la función no modificará los elementos. Si la función necesitara modificar los datos, usaríamos `[]i32` sin const.

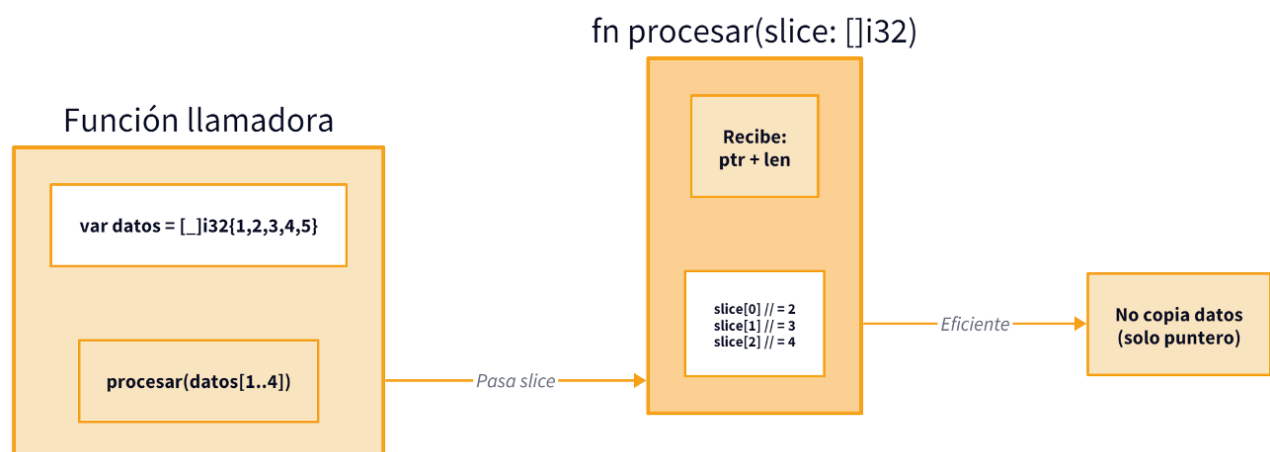


Figura 8 - Slice como parámetro

En Zig, las cadenas de texto (strings) son en realidad slices de bytes. El tipo `[]const u8` representa un string: una secuencia de caracteres (bytes) que no se modificará. Esto explica por qué hemos estado usando este tipo para textos:

1	<code>const mensaje: []const u8 = "¡Hola, Mundo!";</code>
2	
3	<code>// Puedes iterar sobre los caracteres</code>
4	<code>for (mensaje) carácter {</code>
5	<code> std.debug.print("{c}", .{carácter});</code>
6	<code>}</code>
7	
8	<code>// O acceder a caracteres específicos</code>
9	<code>const primera_letra = mensaje[0];</code>

Esta representación unificada significa que todas las técnicas que aprendes para trabajar con slices también aplican a strings: puedes obtener su longitud con `.len`, crear substrings con la sintaxis de rango, y pasarlos a funciones que aceptan `[]const u8`.



Consejo

Cuando trabajes con texto que puede contener caracteres especiales o emojis, recuerda que UTF-8 (la codificación que Zig usa) puede usar múltiples bytes por carácter. Un emoji puede ocupar 4 bytes, así que `mensaje.len` te da el número de bytes, no necesariamente el número de caracteres visibles.

Zig soporta arrays multidimensionales, que son esencialmente arrays de arrays. Son útiles para representar estructuras como matrices, tableros de juego, imágenes, y cualquier dato que tenga múltiples dimensiones:

1	// Matriz de 3 filas x 4 columnas
2	const matriz = [3][4]i32{
3	.{ 1, 2, 3, 4 },
4	.{ 5, 6, 7, 8 },
5	.{ 9, 10, 11, 12 },
6	};

Para acceder a un elemento de una matriz, usas dos índices: primero la fila, luego la columna:

1	const elemento = matriz[1][2]; // Fila 1, Columna 2 = 7
2	
3	// Iterar sobre una matriz
4	for (matriz) fila {
5	for (fila) elemento {
6	std.debug.print("{} ", .{elemento});
7	}
8	std.debug.print("\n", .{ });
9	}

El bucle exterior recorre las filas, y el bucle interior recorre los elementos de cada fila. Este patrón de bucles anidados es muy común cuando trabajas con datos multidimensionales.

Hay varias operaciones que frecuentemente necesitarás realizar con arrays y slices. La librería estándar de Zig proporciona funciones útiles en el módulo std.mem para algunas de estas operaciones:

1	const std = @import("std");
2	
3	const array1 = [_]u8{ 'h', 'o', 'l', 'a' };

4	<code>const array2 = [_]u8{ 'h', 'o', 'l', 'a' };</code>
5	<code>const array3 = [_]u8{ 'a', 'd', 'i', 'o', 's' };</code>
6	
7	<code>// Comparar si dos arrays/slices son iguales</code>
8	<code>const son_iguales = std.mem.eql(u8, &array1, &array2); // true</code>
9	<code>const diferentes = std.mem.eql(u8, &array1, &array3); // false</code>

La función `std.mem.eql` compara dos slices elemento por elemento y retorna `true` si todos los elementos son iguales. Esto es mucho más conveniente que escribir un bucle manual para comparar.

Otra operación común es buscar un elemento dentro de un array:

1	<code>const datos = [_]i32{ 10, 25, 30, 45, 50 };</code>
2	
3	<code>// Buscar la primera ocurrencia de un valor</code>
4	<code>if (std.mem.indexOfScalar(i32, &datos, 30)) indice {</code>
5	<code>std.debug.print("Encontrado en indice {}\n", .{indice});</code>
6	<code>} else {</code>
7	<code>std.debug.print("No encontrado\n", .{});</code>
8	<code>}</code>

La función `indexOfScalar` retorna un valor opcional: si encuentra el elemento, retorna su índice; si no lo encuentra, retorna `null`. El patrón `"if (opcional) |valor|"` es la forma idiomática de manejar estos resultados opcionales en Zig.

Función	Uso	Retorna
<code>std.mem.eql</code>	Comparar dos slices	<code>bool</code>
<code>std.mem.indexOfScalar</code>	Buscar un elemento	<code>?usize</code> (opcional)
<code>std.mem.copy</code>	Copiar elementos	<code>void</code>
<code>std.mem.set</code>	Llenar con un valor	<code>void</code>

Tabla 27 - Funciones Útiles de `std.mem`
(Fuente: Propia)

Zig tiene un concepto especial llamado 'sentinel-terminated arrays', que son arrays que terminan con un valor especial (el sentinel). El ejemplo más común son los strings terminados en null, que son compatibles con código C:

1	// String terminado en null (sentinel 0)
2	const c_string: [:0]const u8 = "Hola";
3	
4	// El tipo [:0]const u8 indica un slice terminado con el byte 0
5	// Esto es compatible con funciones de C que esperan char*

El sentinel se especifica entre los dos puntos y el corchete de cierre. Aunque esto es más relevante cuando trabajas con código C, es bueno conocer el concepto para entender algunos tipos que verás en la librería estándar.

Veamos un programa completo que demuestra varias operaciones con arrays y slices. Este programa analiza las calificaciones de estudiantes:

1	const std = @import("std");
2	
3	fn calcular_promedio(notas: []const f32) f32 {
4	var suma: f32 = 0;
5	for (notas) nota {
6	suma += nota;
7	}
8	return suma / @as(f32, @floatFromInt(notas.len));
9	}
10	
11	fn encontrar_maxima(notas: []const f32) f32 {
12	var max = notas[0];
13	for (notas[1..]) nota {
14	if (nota > max) max = nota;
15	}

16	return max;
17	}
18	
19	fn encontrar_minima(notas: []const f32) f32 {
20	var min = notas[0];
21	for (notas[1..]) nota {
22	if (nota < min) min = nota;
23	}
24	return min;
25	}
26	
27	fn contar_aprobados(notas: []const f32, minimo: f32) u32 {
28	var contador: u32 = 0;
29	for (notas) nota {
30	if (nota ≥ minimo) contador += 1;
31	}
32	return contador;
33	}
34	
35	pub fn main() void {
36	const notas = [_]f32{ 85, 92, 78, 95, 67, 88, 73, 91, 84, 79 };
37	
38	std.debug.print("Análisis de Calificaciones\n", .{});
39	std.debug.print("=====\n", .{});
40	std.debug.print("Total de estudiantes: {}\n", .{notas.len});
41	std.debug.print("Promedio: {d:.1}\n", .{calcular_promedio(¬as)});
42	std.debug.print("Nota maxima: {d:.1}\n", .{encontrar_maxima(¬as)});
43	std.debug.print("Nota minima: {d:.1}\n", .{encontrar_minima(¬as)});

44	<code>std.debug.print("Aprobados (≥ 70): {}\\n", .{contar_aprobados(&notas,</code>
45	<code>70)});</code> <code>}</code>

Video 7 - Arrays y Slices en Zig



Trabajar con arrays y slices frecuentemente involucra algoritmos clásicos de procesamiento de datos. Aunque la librería estándar de Zig proporciona algunas funciones útiles, es valioso entender cómo implementar estos algoritmos tú mismo para comprender mejor el lenguaje.

Algoritmo	Complejidad	Uso típico
Busqueda líneal	O(n)	Encontrar un elemento
Busqueda binaria	O(log n)	Arrays ordenados
Ordenamiento	O(n log n)	Organizar datos
Reversion	O(n)	Invertir orden
Filtrado	O(n)	Seleccionar elementos

Tabla 28 - Algoritmos Comunes en Arrays
(Fuente: Propia)

La búsqueda lineal es el algoritmo más básico: recorres cada elemento hasta encontrar lo que buscas. Es simple pero lento para arrays grandes:

1

```
fn buscar_lineal(slice: []const i32, valor: i32) ?usize {\n    for (slice, 0..) |elemento, indice| {\n        if (elemento == valor) return indice;\n    }\n    return null; // No encontrado\n}\n\nconst números = [_]i32{ 10, 25, 30, 45, 50 };\nconst pos = buscar_lineal(&números, 30); // Retorna 2
```

Para arrays ordenados, la búsqueda binaria es mucho más eficiente. Divide el espacio de búsqueda a la mitad en cada paso:

1

```
fn buscar_binaria(slice: []const i32, valor: i32) ?usize {\n    var izq: usize = 0;\n    var der: usize = slice.len;\n\n    while (izq < der) {\n        const mid = izq + (der - izq) / 2;\n        if (slice[mid] == valor) return mid;\n        if (slice[mid] < valor) {\n            izq = mid + 1;\n        }\n        else {\n            der = mid;\n        }\n    }\n    return null;\n}
```

Invertir un array es útil para muchos problemas. Aquí hay una implementación eficiente que intercambia elementos desde los extremos hacia el centro:

1

```
fn revertir(slice: []i32) void {\n    if (slice.len == 0) return;\n    var izq: usize = 0;\n    var der: usize = slice.len - 1;\n\n    while (izq < der) {\n        const temp = slice[izq];\n        slice[izq] = slice[der];\n        slice[der] = temp;\n        izq += 1;\n        der -= 1;\n    }\n}
```

Al trabajar con arrays y slices, ciertos patrones aparecen repetidamente. Reconocer estos patrones te ayudará a escribir código más claro y eficiente.

Patrón	Descripción	Ejemplo
Acumulación	Reducir a un valor	Suma, producto, máximo
Transformación	Modificar cada elemento	Multiplicar por 2
Filtrado	Seleccionar elementos	Mayores que X
Mapeo	Convertir elementos	Entero a string
Ventana deslizante	Procesar subgrupos	Promedio movil

Tabla 29 - Patrones de Uso de Arrays
(Fuente: Propia)

El patrón de acumulación reduce un array a un solo valor. Ya vimos ejemplos con suma y promedio. Aquí hay una versión más genérica usando comptime:

```
1 fn reducir(comptime T: type, slice: []const T, inicial: T, func: fn(T, T) T)
  T {\n    var resultado = inicial;\n    for (slice) |elem| {\n
resultado = func(resultado, elem);\n    }\n    return resultado;\n}\n\nfn
sumar(a: i32, b: i32) i32 { return a + b; }\nfn maximo(a: i32, b: i32) i32 {
return if (a > b) a else b; }\n\nconst nums = [_]i32{ 5, 3, 8, 1, 9 };\n
const suma = reducir(i32, &nums, 0, sumar); // 26\nconst max =
reducir(i32, &nums, nums[0], maximo); // 9
```

El patrón de ventana deslizante es útil para calcular estadísticas sobre subgrupos consecutivos, como promedios móviles:

```
1 // Calcular promedio móvil de tamaño 3\nfn promedio_movil(datos: []const
f32, resultado: []f32) void {\n    const ventana = 3;\n    for (0..
(datos.len - ventana + 1)) |i| {\n        var suma: f32 = 0;\n        for
(datos[i..i+ventana]) |v| suma += v;\n        resultado[i] = suma / ventana;\n
    }\n}
```



Consejo

Cuando trabajes con grandes volúmenes de datos, considera usar la librería estándar de Zig que incluye algoritmos optimizados. `std.mem` y `std.sort` ofrecen funciones eficientes para ordenamiento y búsqueda.

En este capítulo aprendiste sobre arrays y slices, las estructuras de datos fundamentales para trabajar con colecciones de elementos. Los arrays son colecciones de tamaño fijo, donde el tamaño es parte del tipo y se conoce en tiempo de compilación. Los slices son vistas flexibles sobre secuencias de datos, con tamaño determinado en tiempo de ejecución.

Viste cómo crear arrays con valores iniciales, cómo acceder a elementos por índice, y cómo iterar sobre todos los elementos usando bucles `for`. Aprendiste que Zig verifica automáticamente los límites de los arrays para prevenir errores de acceso inválido.

Exploramos los slices y su flexibilidad para trabajar con funciones, ya que una función que acepta un slice puede trabajar con arrays de cualquier tamaño. También viste que los strings

en Zig son simplemente slices de bytes (`[]const u8`), lo que permite aplicar las mismas técnicas a texto.

EJERCICIOS

Pon a prueba tus conocimientos con los siguientes ejercicios:

SUMA Y PROMEDIO

Crea un array con las temperaturas de una semana (7 valores flotantes).
Escribe funciones para calcular la suma total y el promedio de las temperaturas. Muestra ambos resultados.

Ejercicio

1

INVERTIR ARRAY

Escribe una función que reciba un array mutable de enteros y lo invierta en su lugar (el primer elemento se convierte en el último, etc.). Prueba la función con un array de 5 elementos.

Ejercicio

2

CAPÍTULO 8

Punteros y Memoria

Cada vez que creas una variable en tu programa, esa variable ocupa un espacio en la memoria de la computadora. Ese espacio tiene una ubicación específica, cómo una casa tiene una dirección postal. Los punteros son variables especiales que, en lugar de almacenar un valor como un número o un texto, almacenan la dirección de memoria donde se encuentra otro valor. Entender los punteros es fundamental para comprender cómo funcionan los programas a bajo nivel y para aprovechar al máximo las capacidades de Zig.

Imagina que tienes un amigo que te pregunta dónde vive tu primo. Puedes responder de dos formas: podrías describir la casa con todo detalle (el color, las ventanas, el jardín), o simplemente podrías darle la dirección. La dirección es mucho más eficiente: es corta, fácil de comunicar, y permite a tu amigo llegar directamente a la casa. Los punteros funcionan de manera similar: en lugar de copiar todos los datos, simplemente guardas la dirección dónde se encuentran.

Los punteros son especialmente importantes en Zig porque el lenguaje está diseñado para programación de sistemas, donde el control preciso sobre la memoria es esencial. A diferencia de lenguajes de alto nivel que ocultan estos detalles, Zig te da acceso directo a la memoria mientras te proporciona herramientas para hacerlo de forma segura. En este capítulo aprenderás qué son los punteros, cómo usarlos, y las reglas que Zig establece para prevenir errores comunes.

Antes de hablar de punteros, necesitamos entender qué es una dirección de memoria. La memoria de una computadora es como una larga fila de casilleros numerados. Cada casillero puede almacenar un byte de información, y cada uno tiene un número único que lo identifica: su dirección. Cuando creas una variable, el compilador reserva uno o más de estos casilleros para almacenar su valor.

En Zig, puedes obtener la dirección de una variable usando el operador ampersand (&). Este operador te da un puntero a la variable:

El tipo de ptr es `*i32`, que se lee como "puntero a i32". Esto significa que ptr contiene la

1	<code>var número: i32 = 42;</code>
2	<code>const ptr = &número; // ptr contiene la dirección de 'número'</code>
3	
4	<code>debug.print("Valor: {}\n", .{número});</code>
5	<code>debug.print("Dirección: {}\n", .{ptr});</code>

dirección de un valor de tipo i32. El asterisco (*) antes del tipo indica que es un puntero.

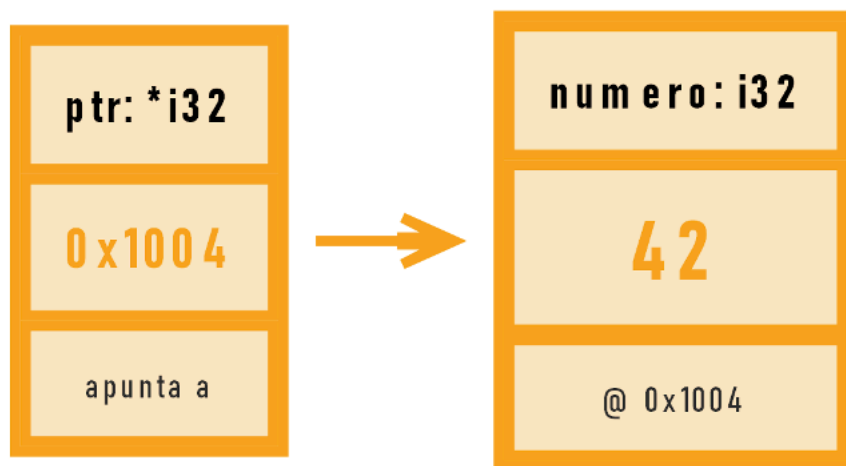


Figura 9 - Diagrama de puntero ptr

Zig tiene varios tipos de punteros, cada uno con diferentes características. El más básico es el puntero simple (`*T`), que apunta a exactamente un valor de tipo `T`. También existe el puntero a muchos elementos (`[*]T`), que apunta a una secuencia de valores, y el puntero con sentinel (`[*:x]T`), que apunta a una secuencia terminada con un valor especial.

1	<code>var num: i32 = 100;</code>
2	<code>const ptr_simple: *i32 = &num; // Puntero a un solo i32</code>
3	
4	<code>var array = [_]i32{ 1, 2, 3, 4, 5 };</code>
5	<code>const ptr_muchos: [*]i32 = &array; // Puntero a varios i32</code>

Tipo	Descripción	Uso
*T	Puntero a un valor	Un solo elemento
[*]T	Puntero a varios valores	Secuencia de elementos
?*T	Puntero opcional	Puede ser null
*const T	Puntero a valor constante	Solo lectura

Tabla 30 - Tipos de Punteros en Zig
(Fuente: Propia)

Además, los punteros pueden ser const o no const. Un puntero *const T apunta a un valor que no puede ser modificado a través del puntero. Un puntero *T (sin const) permite modificar el valor al que apunta. Esta distinción es importante para expresar tus intenciones y para que el compilador pueda verificar que no modificas datos que deberían ser inmutables.

Una vez que tienes un puntero, probablemente quieras acceder al valor al que apunta. Esta operación se llama "dereferenciación" y se realiza con el operador punto-asterisco (.*). Dereferenciar un puntero te da el valor que está almacenado en la dirección a la que apunta:

1	var número: i32 = 42;
2	const ptr = &número;
3	
4	// Leer el valor a través del puntero
5	const valor = ptr.*; // valor es 42
6	
7	std.debug.print("Valor leído del puntero: {}\n", .{valor});

Operador	Uso	Ejemplo
&	Obtener dirección	&variable → *T
.*	Dereferenciar (leer/escribir)	ptr.* = 10

Tabla 31 - Operadores de Punteros
(Fuente: Propia)

También puedes usar la dereferenciación para modificar el valor al que apunta el puntero, siempre y cuando el puntero no sea const y la variable original sea mutable:

1	<code>var número: i32 = 42;</code>
2	<code>const ptr = &número;</code>
3	
4	<code>// Modificar el valor a través del puntero</code>
5	<code>ptr.* = 100;</code>
6	
7	<code>std.debug.print("Nuevo valor: {}\n", .{número}); // Imprime 100</code>

Esto es muy poderoso porque nos permite que una función modifique variables que están fuera de ella. Recuerda que en el capítulo de funciones vimos que los parámetros se pasan por valor (se copian). Con punteros, podemos pasar la dirección de una variable, permitiendo que la función modifique el original.



NOTA

En otros lenguajes como C, la direferenciación usa el prefijo `*ptr`. En Zig, usamos el sufijo `ptr.*` para ser consistentes con la sintaxis de acceso a campos. Esto hace el código más legible cuando encadenas operaciones.

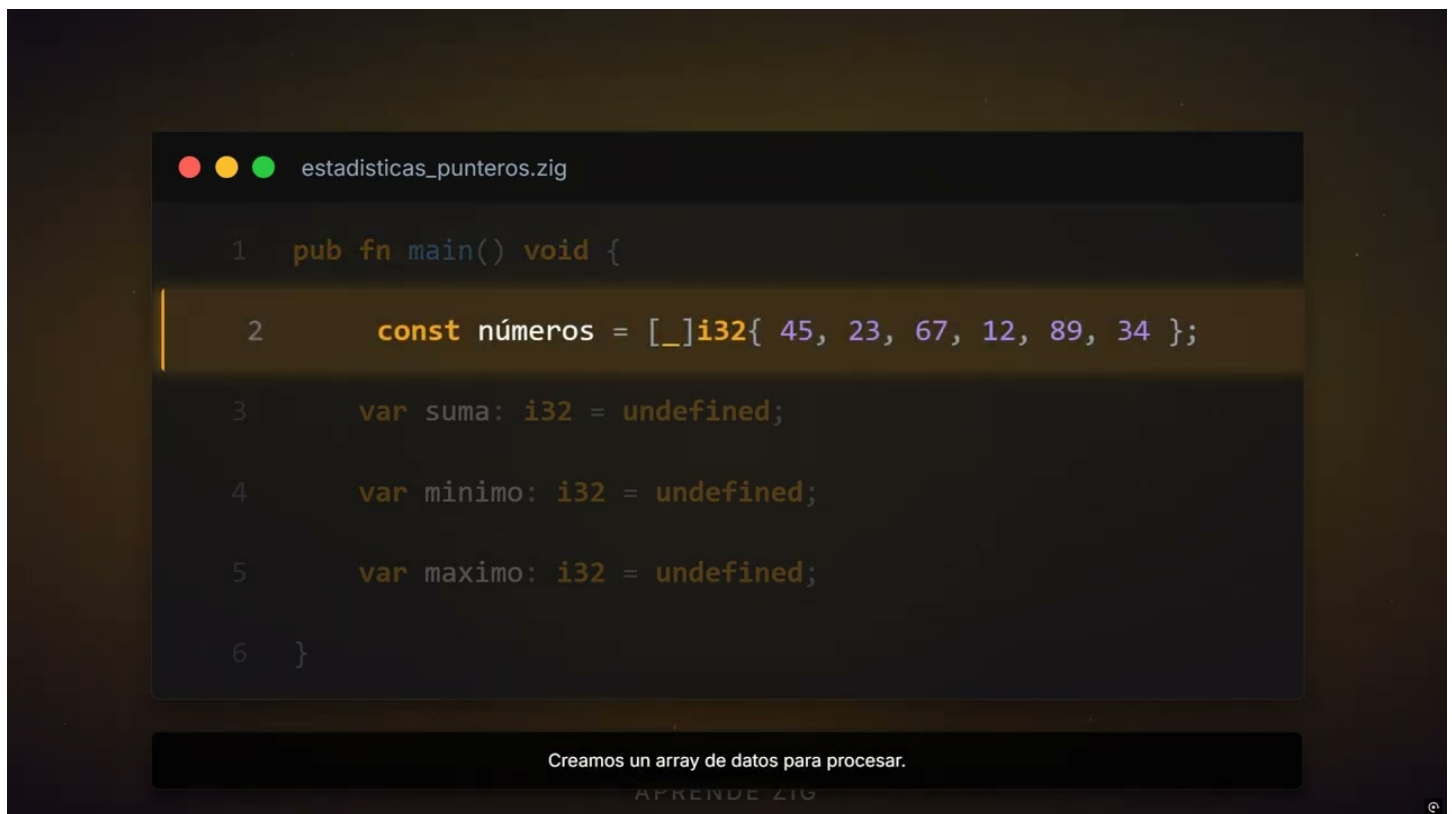
Uno de los usos más importantes de los punteros es permitir que las funciones modifiquen datos externos. Cuando pasas un puntero a una función, la función puede acceder y modificar el valor original:

1	fn duplicar(ptr: *i32) void {
2	ptr.* = ptr.* * 2; // Modifica el valor original
3	}
4	
5	fn incrementar(ptr: *i32) void {
6	ptr.* += 1;
7	}
8	
9	pub fn main() void {
10	var número: i32 = 10;
11	
12	std.debug.print("Original: {}\n", .{número}); // 10
13	duplicar(&número);
14	std.debug.print("Duplicado: {}\n", .{número}); // 20
15	incrementar(&número);
16	std.debug.print("Incrementado: {}\n", .{número}); // 21
17	}

Observa que pasamos &número (la dirección de número) a las funciones, y las funciones reciben parámetros de tipo *i32 (puntero a i32). Dentro de la función, usamos ptr.* para acceder y modificar el valor. Los cambios persisten después de que la función retorna porque estamos modificando la variable original, no una copia.

Esta técnica es útil cuando una función necesita modificar múltiples valores, ya que solo puede retornar un valor directamente. Con punteros, puede modificar tantos valores como punteros reciba:

1	fn intercambiar(a: *i32, b: *i32) void {
2	const temp = a.*;
3	a.* = b.*;
4	b.* = temp;
5	}
6	
7	pub fn main() void {
8	var x: i32 = 5;
9	var y: i32 = 10;
10	
11	std.debug.print("Antes: x={}, y={}\\n", .{x, y});
12	intercambiar(&x, &y);
13	std.debug.print("Después: x={}, y={}\\n", .{x, y});
14	}



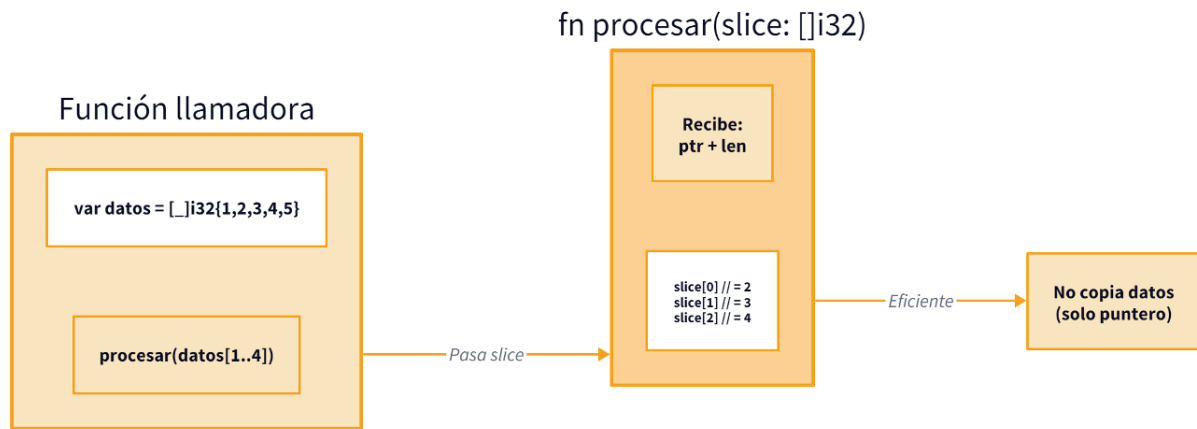


Figura 10 - Puntero en función

En el capítulo anterior vimos que los slices son vistas flexibles sobre arrays. Internamente, un slice es una combinación de un puntero al primer elemento y un contador de longitud. Cuando pasas un slice a una función, estás pasando esta información, lo que permite a la función acceder a los datos originales.

1	fn sumar_uno_a_todos(números: []i32) void {
2	for (números) *n { // *n captura un puntero a cada elemento
3	n.* += 1;
4	}
5	}
6	
7	pub fn main() void {
8	var datos = []i32{ 1, 2, 3, 4, 5 };
9	
10	std.debug.print("Antes: ", .{});
11	for (datos) n std.debug.print("{} ", .{n});
12	
13	sumar_uno_a_todos(&datos);
14	
15	std.debug.print("\nDespués: ", .{});
16	for (datos) n std.debug.print("{} ", .{n});
17	}

Observa la sintaxis `|*nl` en el bucle `for`. Normalmente, `lnl` captura el valor de cada elemento (una copia). Al usar `|*nl`, capturas un puntero al elemento, lo que te permite modificar el array original. Este es un patrón muy común en Zig.

En muchos lenguajes, los punteros pueden ser "null", indicando que no apuntan a ningún lugar válido. Los punteros nulos son una fuente frecuente de errores graves (los famosos "null pointer exceptions"). Zig aborda este problema de manera diferente: los punteros regulares (`*T`) nunca pueden ser nulos.

Si necesitas un puntero que pueda ser nulo, debes usar un puntero opcional (`?*T`). El signo de interrogación indica que el puntero puede contener una dirección válida o puede ser null:

1	<code>fn buscar_elemento(datos: []const i32, valor: i32) ?*const i32 {</code>
2	<code> for (datos) *elem {</code>
3	<code> if (elem.* == valor) {</code>
4	<code> return elem; // Retorna puntero al elemento encontrado</code>
5	<code> }</code>
6	<code> }</code>
7	<code> return null; // No se encontró</code>
8	<code>}</code>

Cuando trabajas con punteros opcionales, debes verificar si contienen un valor antes de desreferenciarlos. Zig te obliga a hacer esta verificación, previniendo accesos a punteros nulos:

1	<code>const datos = [_]i32{ 10, 20, 30, 40, 50 };</code>
2	
3	<code>if (buscar_elemento(&datos, 30)) ptr {</code>
4	<code> std.debug.print("Encontrado: {}\n", .{ptr.*});</code>
5	<code>} else {</code>
6	<code> std.debug.print("No encontrado\n", .{});</code>
7	<code>}</code>



Consejo

La regla general es: usa punteros regulares (*T) cuando sabes que el puntero siempre será válido. Usa punteros opcionales (?*T) cuando el puntero podría no existir (como cuando buscas algo que podría no encontrarse). Esto hace tus intenciones claras y permite que el compilador te ayude a prevenir errores.

Carácterística	*T (Regular)	?*T (Opcional)
Puede ser null	No	Si
Verificación requerida	No	Si
Uso típico	Siempre válido	Puede no existir

Tabla 32 - Punteros Regulares vs Opcionales
(Fuente: Propia)

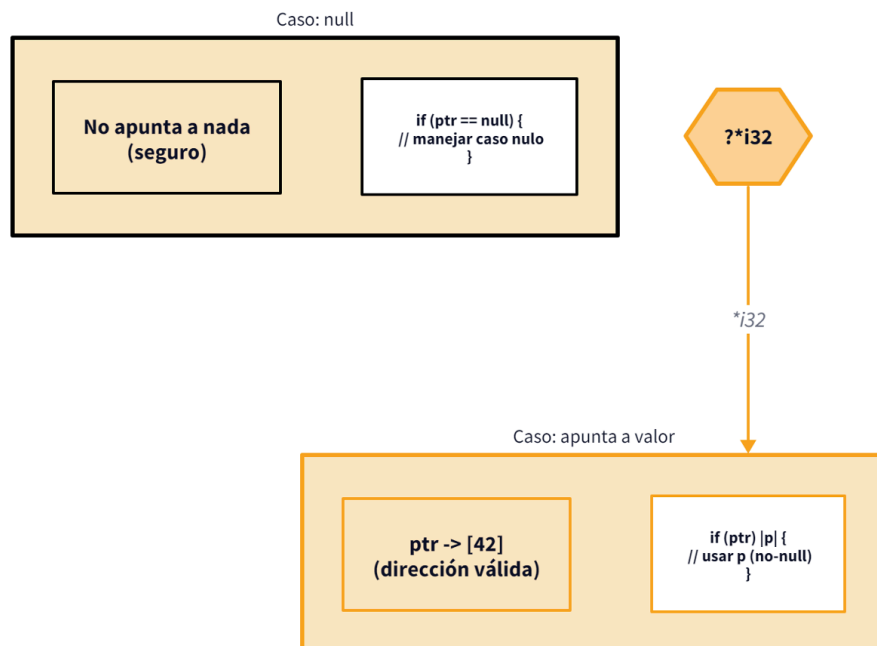


Figura 11 - Puntero opcional

Hasta ahora, todas las variables que hemos creado viven en lo que se llama el "stack" (pila). El stack es una región de memoria que se maneja automáticamente: cuando una función comienza, se reserva espacio para sus variables locales, y cuando la función termina, ese espacio se libera automáticamente. Esto es eficiente, pero tiene limitaciones: las variables del stack tienen un tamaño fijo conocido en tiempo de compilación.

Para datos que necesitan un tamaño dinámico o que deben sobrevivir más allá de la función que los crea, existe el "heap" (montículo). El heap es una región de memoria donde puedes reservar espacio manualmente cuando lo necesites y liberarlo cuando ya no lo uses. Zig proporciona allocators para manejar la memoria del heap.

Carácterística	Stack	Heap
Gestión	Automática	Manual (allocators)
Tamaño	Fijo en compilación	Dinámico
Vida útil	Duración del bloque	Controlada por ti
Velocidad	Muy rápido	Más lento

Tabla 33 - Stack vs Heap
(Fuente: Propia)

1	<code>const std = @import("std");</code>
2	
3	<code>pub fn main() !void {</code>
4	<code> // Obtener un allocator</code>
5	<code> var gpa = std.heap.GeneralPurposeAllocator(.{}){};</code>
6	<code> defer _ = gpa.deinit();</code>
7	<code> const allocator = gpa.allocator();</code>
8	
9	<code> // Reservar memoria para 10 enteros en el heap</code>
10	<code> const datos = try allocator.alloc(i32, 10);</code>
11	<code> defer allocator.free(datos); // Liberar al salir</code>
12	

13	<code>// Usar los datos</code>
14	<code>for (datos, 0..) *ptr, i {</code>
15	<code> ptr.* = @intCast(i * 10);</code>
16	<code>}</code>
17	
18	<code>for (datos) n {</code>
19	<code> std.debug.print("{} ", .{n});</code>
20	<code>}</code>
21	<code>}</code>

Este ejemplo introduce varios conceptos nuevos. El `GeneralPurposeAllocator` es un tipo de allocator que Zig proporciona para propósitos generales. La función `alloc` reserva memoria y retorna un slice que apunta a ella. La palabra clave `defer` asegura que la memoria se libere cuando el bloque actual termine, incluso si ocurre un error. Esto previene fugas de memoria.

NOTA

El manejo de memoria del heap requiere cuidado. Si reservas memoria, pero olvidas liberarla, tienes una 'fuga de memoria'. Si liberas memoria y luego intentas usarla, tienes un 'use after free'. Zig te ayuda con herramientas como `defer` para hacer el manejo de memoria más seguro, pero la responsabilidad final es tuya.

La palabra clave `defer` es una herramienta poderosa para manejar recursos. Cuando escribes una sentencia `defer`, estás diciendo: "ejecuta este código cuando salga del bloque actual, sin importar como salga". Esto incluye retornos normales, retornos tempranos, y errores.

1	<code>fn procesar_archivo() !void {</code>
2	<code> var archivo = try abrir_archivo();</code>
3	<code> defer cerrar_archivo(archivo); // Se ejecutara al final</code>
4	
5	<code> // Procesar archivo...</code>
6	<code> // No importa como salga la función, el archivo se cerrara</code>
7	<code>}</code>

Sin defer, tendrías que asegurarte de cerrar el archivo en cada punto donde la función podría terminar, lo cual es tedioso y propenso a errores. Con defer, el código de limpieza se escribe una sola vez, justo después de adquirir el recurso, y se ejecutará automáticamente.

Puedes tener múltiples defers en una función. Se ejecutan en orden inverso al que fueron declarados (el último defer declarado se ejecuta primero). Esto es útil cuando los recursos tienen dependencias entre si.

Zig está diseñado para ser un lenguaje seguro sin sacrificar el rendimiento. Varias características del sistema de punteros ayudan a prevenir errores comunes:

Primera, los punteros regulares nunca son nulos. Si un puntero existe, siempre apunta a datos válidos. Los punteros opcionales (?*T) hacen explícito cuando un puntero podría no existir.

Segunda, Zig no tiene aritmética de punteros implícita. En C, puedes sumar un número a un puntero para moverte en la memoria. En Zig, esto requiere convertir explícitamente el puntero, haciendo estas operaciones más visibles y deliberadas.

Tercera, los slices incluyen información de longitud. Cuando pasas un slice, la función receptora sabe exactamente cuántos elementos hay, previniendo accesos fuera de límites.

1	<code>const std = @import("std");</code>
2	
3	<code>fn calcular_estadisticas(</code>
4	<code> datos: []const i32,</code>
5	<code> out_suma: *i32,</code>
6	<code> out_min: *i32,</code>
7	<code> out_max: *i32,</code>
8	<code>) void {</code>
9	<code> var suma: i32 = 0;</code>
10	<code> var min = datos[0];</code>
11	<code> var max = datos[0];</code>
12	
13	<code> for (datos) n {</code>
14	<code> suma += n;</code>
15	<code> if (n < min) min = n;</code>
16	<code> if (n > max) max = n;</code>
17	<code> }</code>
18	
19	<code> out_suma.* = suma;</code>
20	<code> out_min.* = min;</code>
21	<code> out_max.* = max;</code>
22	<code>}</code>
23	
24	<code>pub fn main() void {</code>
25	<code> const números = [_]i32{ 45, 23, 67, 12, 89, 34 };</code>
26	
27	<code> var suma: i32 = undefined;</code>
28	<code> var minimo: i32 = undefined;</code>

29	<code>var maximo: i32 = undefined;</code>
30	
31	<code>calcular_estadisticas(&números, &suma, &minimo, &maximo);</code>
32	
33	<code>std.debug.print("Suma: {}\n", .{suma});</code>
34	<code>std.debug.print("Minimo: {}\n", .{minimo});</code>
35	<code>std.debug.print("Maximo: {}\n", .{maximo});</code>
36	<code>}</code>

Un concepto avanzado pero importante es la alineación de memoria. Los procesadores modernos acceden a la memoria de forma más eficiente cuando los datos están alineados a direcciones específicas. Por ejemplo, un entero de 4 bytes se accede más rápido si está en una dirección divisible por 4.

Tipo	Tamaño (Bytes)	Alineación
u8, i8	1	1 byte
u16, i16	2	2 bytes
u32, i32, f32	4	4 bytes
u64, i64, f64	8	8 bytes
usize, punteros	8 (64-bit)	8 bytes

Tabla 34 - Alineación por Tipo
(Fuente: Propia)

Zig maneja la alineación automáticamente en la mayoría de los casos, pero te permite controlarla cuando es necesario. Por ejemplo, al interoperar con hardware o bibliotecas C que requieren alineaciones específicas:

1	<code>// Forzar alineación específica para interoperabilidad\nconst AlignedData = extern struct {\n x: u32 align(16), // Alíneado a 16 bytes\n y: u32,\n};\n\n// Obtener información de alineación\nconst alignment = @alignOf(i32); // Retorna 4</code>
---	--

La función `@alignOf` te dice la alineación natural de un tipo, y `@alignCast` te permite convertir un puntero a otra alineación (verificando en runtime en modo debug). Esto es útil para operaciones SIMD y optimizaciones de bajo nivel.

Los errores de memoria son notoriamente difíciles de depurar. Un programa puede parecer funcionar correctamente y fallar horas después debido a corrupción de memoria. Zig proporciona varias herramientas para detectar estos problemas tempranamente.

Herramienta	Propósito	Cuando Usar
<code>std.testing.allocat or</code>	Detecta fugas	En tests unitarios
<code>GeneralPurposeA llocator</code>	Detecta double-free y use-after-free	En desarrollo
<code>@setRuntimeSafet y</code>	Activas verificaciones extra	Debug builds
Valgrind/ AddressSanitizer	Análisis externo	Problemas complejos

Tabla 35 - Herramientas de Debugging de Memoria
(Fuente: Propia)

El `GeneralPurposeAllocator` (GPA) es especialmente útil durante el desarrollo. Detecta errores comunes y proporciona mensajes claros sobre que salió mal:

1	<pre>const std = @import(\"std\");\n\npub fn main() void {\n var gpa =\nstd.heap.GeneralPurposeAllocator(.{\n .safety = true, // Activa\n deteccion de errores\n }){};\n defer _ = gpa.deinit(); // Reporta\n fugas al terminar\n\n const allocator = gpa.allocator();\n\n const\nptr = allocator.alloc(u8, 100) catch return;\n // Si olvidamos liberar\n'ptr', gpa.deinit() lo reportara\n allocator.free(ptr); // Correcto\n}</pre>
---	--



Consejo

Siempre usa `GeneralPurposeAllocator` durante el desarrollo. El pequeño costo de rendimiento vale la pena por la detección temprana de errores. Puedes cambiar a un allocator más rápido en producción.

Cuando Zig detecta un error de memoria, proporciona un stack trace que te dice exactamente donde se asignó la memoria problemática. Esto es invaluable comparado con lenguajes donde los errores de memoria causan crashes misteriosos.

Si vienes de otros lenguajes, es útil comparar cómo manejan la memoria. Cada enfoque tiene ventajas y desventajas, y entenderlos te ayudara a apreciar las decisiones de diseño de Zig. Si no conoces estos lenguajes, puedes saltarte esta tabla sin perder continuidad.

Lenguaje	Modelo	Características Principal
C	Manual	Sin protecciones, errores catastróficos
C++	Manual + RAII	Smart pointers, complejidad alta
Rust	Ownership	Seguro, pero curva de aprendizaje alta
Go/Java	Garbage Collection	Automático, pero con pausas
Zig	Manual + Allocators	Explicito, seguro con herramientas

Tabla 36 - Manejo de Memoria: Zig vs Otros
(Fuente: Propia)

En C, los punteros son completamente libres: puedes sumar, restar, convertir, y dereferenciar cualquier puntero sin verificación. Esto da máximo rendimiento, pero también

máximo riesgo. Los errores de memoria en C causan vulnerabilidades de seguridad como buffer overflows.

C++ añade smart pointers (`unique_ptr`, `shared_ptr`) que manejan memoria automáticamente, pero son opcionales: puedes mezclar código seguro e inseguro. Rust va más lejos con su sistema de ownership que garantiza seguridad en tiempo de compilación, pero tiene una curva de aprendizaje pronunciada.

Zig toma un enfoque intermedio: el manejo de memoria es explícito (como C) pero con herramientas que lo hacen más seguro. Los allocators se pasan explícitamente, defer asegura limpieza, y el `GeneralPurposeAllocator` detecta errores durante el desarrollo. No hay magia oculta, pero tampoco estás solo contra los bugs.



En este capítulo exploramos los punteros y el manejo de memoria en Zig. Aprendiste que un puntero es una variable que contiene la dirección de memoria de otro valor. El operador `&` obtiene la dirección de una variable, y el operador `*` de referencia un puntero para acceder al valor al que apunta.

Viste cómo los punteros permiten que las funciones modifiquen valores externos, superando la limitación de solo poder retornar un valor. También aprendiste sobre los punteros opcionales (`?*T`) que pueden ser nulos, y cómo Zig te obliga a verificar su validez antes de usarlos.

Exploramos la diferencia entre memoria stack (automática, de tamaño fijo) y heap (manual, dinámica), y cómo los allocators de Zig te ayudan a manejar la memoria del heap. La palabra clave `defer` es una herramienta esencial para asegurar que los recursos se liberen correctamente.

En el próximo capítulo aprenderemos sobre estructuras y tipos personalizados, que te permitirán agrupar datos relacionados en tipos con significado para tu programa.

EJERCICIOS

Pon a prueba tus conocimientos con los siguientes ejercicios

INTERCAMBIO CON PUNTEROS

Escribe una función 'intercambiar' que reciba punteros a dos variables `i32` e intercambie sus valores. Prueba la función con dos variables.

Ejercicio

MODIFICAR ARRAY EN FUNCIÓN

Escribe una función 'duplicar_valores' que reciba un slice de enteros y duplique el valor de cada elemento. Prueba con un array de al menos 5 elementos.

Ejercicio

CAPÍTULO 9

Estructuras y Tipos Personalizados

Hasta ahora hemos trabajado con tipos de datos primitivos: enteros, flotantes, booleanos, y arrays. Estos tipos son los bloques de construcción básicos, pero para modelar problemas del mundo real, necesitamos formas de combinarlos en tipos más complejos y significativos. Imagina que estás escribiendo un programa para gestionar una biblioteca. Cada libro tiene un título, un autor, un número de páginas, y un año de publicación. Podrías usar variables separadas para cada atributo, pero esto se vuelve caótico rápidamente. ¿No sería mejor tener un tipo 'Libro' que contenga toda esta información de forma organizada?

Las estructuras (o structs) te permiten crear exactamente eso: tipos personalizados que agrupan varios valores relacionados bajo un solo nombre.

Tipo	Uso	Ejemplo
struct	Agrupar datos relacionados	Punto, Libro, Persona
enum	Valores de conjunto fijo	DiaSemana, Estado
Unión	Diferentes tipos alternativos	Mensaje, Resultado

Tabla 37 - Tipos Personalizados en Zip
(Fuente: Propia)

Libro	
titulo	[]const u8
autor	[]const u8
paginas	u16
anio	u16

Figura 12 - Diagrama de struct Libro

Una estructura en Zig se define con la palabra clave 'struct' seguida de los campos que contiene. Cada campo tiene un nombre y un tipo. Veamos cómo definir una estructura simple para representar un punto en dos dimensiones:

1	const Punto = struct {
2	x: f32,
3	y: f32,
4	};

Hemos creado un nuevo tipo llamado 'Punto' que contiene dos campos: 'x' e 'y', ambos de tipo f32. Ahora podemos crear variables de tipo Punto e inicializarlas con valores específicos para cada campo:

1	const origen = Punto{ .x = 0, .y = 0 };
2	const posicion = Punto{ .x = 3.5, .y = 7.2 };
3	
4	std.debug.print("Posicion: ({d}, {d})\n", .{posicion.x, posicion.y});

Observa la sintaxis de inicialización: usamos el nombre del tipo seguido de llaves, y dentro especificamos cada campo precedido por un punto. Esto se llama sintaxis de inicializador designado. Para acceder a los campos de una estructura, usamos la notación de punto: posicion.x nos da el valor del campo x.

Las estructuras pueden tener tantos campos como necesites, y los campos pueden ser de cualquier tipo, incluyendo otras estructuras:

1	const Libro = struct {
2	titulo: []const u8,
3	autor: []const u8,
4	páginas: u32,
5	anio: u16,
6	disponible: bool,
7	};
8	
9	const mi_libro = Libro{
10	.titulo = "El Principito",
11	.autor = "Antoine de Saint-Exupery",
12	.páginas = 96,
13	.anio = 1943,
14	.disponible = true,
15	};

A veces quieres que ciertos campos tengan un valor predeterminado si no se especifican durante la inicialización. Zig permite definir valores por defecto para los campos de una estructura:

1	const Configuracion = struct {
2	puerto: u16 = 8080,
3	max_conexiones: u32 = 100,
4	debug: bool = false,
5	nombre: []const u8 = "servidor",
6	};
7	
8	// Crear con todos los valores por defecto
9	const config_default = Configuracion{};

10	
11	// Crear sobrescribiendo solo algunos valores
12	const config_custom = Configuracion{
13	.puerto = 3000,
14	.debug = true,
15	};

En config_custom, solo especificamos puerto y debug, así que max_conexiones y nombre mantienen sus valores por defecto. Esto es muy útil para configuraciones donde la mayoría de los valores raramente cambian.

Las estructuras en Zig pueden contener no solo datos sino también funciones relacionadas. Estas funciones se llaman métodos y operan sobre la propia estructura.

1	const Rectángulo = struct {
2	ancho: f32,
3	alto: f32,
4	pub fn area(self: Rectángulo) f32 {
5	return self.ancho * self.alto;
6	}
7	
8	pub fn perimetro(self: Rectángulo) f32 {
9	return 2 * (self.ancho + self.alto);
10	}
11	
12	pub fn es_cuadrado(self: Rectángulo) bool {
13	return self.ancho == self.alto;
14	}
15	};

Los métodos reciben como primer parámetro una referencia a la instancia de la estructura, convencionalmente llamada 'self'. Esto les permite acceder a los campos de la estructura. Para llamar a un método, usamos la misma notación de punto que para acceder a campos:

1	<code>const rect = Rectángulo{ .ancho = 5, .alto = 3 };</code>
2	
3	<code>std.debug.print("Area: {d}\n", .{rect.area()}); // 15</code>
4	<code>std.debug.print("Perimetro: {d}\n", .{rect.perimetro()}); // 16</code>
5	<code>std.debug.print("¿Es cuadrado?: {}\n", .{rect.es_cuadrado()}); // false</code>

Cuando el método necesita modificar la estructura, el primer parámetro debe ser un puntero en lugar de un valor. Esto permite que los cambios persistan después de que el método retorne:

1	<code>const Contador = struct {</code>
2	<code> valor: i32,</code>
3	
4	<code> pub fn incrementar(self: *Contador) void {</code>
5	<code> self.valor += 1;</code>
6	<code> }</code>
7	
8	<code> pub fn decrementar(self: *Contador) void {</code>
9	<code> self.valor -= 1;</code>
10	<code> }</code>
11	
12	<code> pub fn reset(self: *Contador) void {</code>
13	<code> self.valor = 0;</code>
14	<code> }</code>
15	<code>};</code>
16	
17	<code>var contador = Contador{ .valor = 10 };</code>
18	<code>contador.incrementar();</code>

NOTA

Cuando llamas a un método que espera un puntero (*Contador), Zig automáticamente toma la referencia de la variable. No necesitas escribir (&contador).incrementar(), simplemente contador.incrementar() funciona si contador es una variable mutable.

Parámetro	Puede modificar	Uso
self: T	No	Solo lectura (area, mostrar)
self: *T	Si	Modificar campos (incrementar, reset)

Tabla 38 - Parámetro self en Métodos
(Fuente: Propia)

Las enumeraciones (enums) son perfectas cuando un valor solo puede ser una de un conjunto fijo de opciones. Piensa en los días de la semana, los estados de un pedido, o los colores de un semáforo: hay un número limitado y conocido de posibilidades.

1	const DiaSemana = enum {
2	lunes,
3	martes,
4	miercoles,
5	jueves,

6	viernes,
7	sabado,
8	domingo,
9	};

Cada opción dentro de un enum se llama una variante. Para usar una variante, la precedemos con un punto. Las enumeraciones funcionan muy bien con la sentencia switch, que puede verificar que hayas manejado todas las variantes:

1	const hoy = DiaSemana.viernes;
2	
3	const mensaje = switch (hoy) {
4	.lunes, .martes, .miercoles, .jueves ⇒ "Dia laboral",
5	.viernes ⇒ "¡Casi fin de semana!",
6	.sabado, .domingo ⇒ "Fin de semana",
7	};
8	
9	std.debug.print("{s}\n", .{mensaje});

Si olvidas manejar alguna variante en el switch, el compilador te lo advertirá. Esto es una característica de seguridad que previene errores sutiles donde accidentalmente ignoras un caso.

Los enums también pueden tener métodos, igual que las estructuras:

1	const Estado = enum {
2	pendiente,
3	procesando,
4	completado,
5	error,
6	

7	<code>pub fn es_final(self: Estado) bool {</code>
8	<code> return self == .completado or self == .error;</code>
9	<code>}</code>
10	
11	<code>pub fn descripción(self: Estado) []const u8 {</code>
12	<code> return switch (self) {</code>
13	<code> .pendiente => "Esperando inicio",</code>
14	<code> .procesando => "En progreso",</code>
15	<code> .completado => "Finalizado con éxito",</code>
16	<code> .error => "Fallo",</code>
17	<code> };</code>
18	<code>}</code>
19	<code>};</code>



Consejo

Para imprimir el nombre de una variante de enum tal cual (por ejemplo 'rojo' o 'verde'), puedes usar la función incorporada `@tagName()`. Es muy útil para depuración: `std.debug.print("{s}", .{@tagName(mi_enum)})`;

Las uniones (unions) son tipos que pueden almacenar diferentes tipos de valores, pero solo uno a la vez. Son útiles cuando un dato puede tener diferentes formas.

1	const Número = unión(enum) {
2	entero: i64,
3	flotante: f64,
4	sin_valor: void,
5	};
6	
7	const a = Número{ .entero = 42 };
8	const b = Número{ .flotante = 3.14 };
9	const c = Número{ .sin_valor = {} };

Para acceder al valor de una unión, debes usar switch, lo cual garantiza que manejes correctamente cada posible tipo de contenido:

1	fn mostrar_número(n: Número) void {
2	switch (n) {
3	.entero => valor std.debug.print("Entero: {}\n", .{valor}),
4	.flotante => valor std.debug.print("Flotante: {d}\n", .{valor}),
5	.sin_valor => std.debug.print("Sin valor\n", .{}),
6	}
7	}

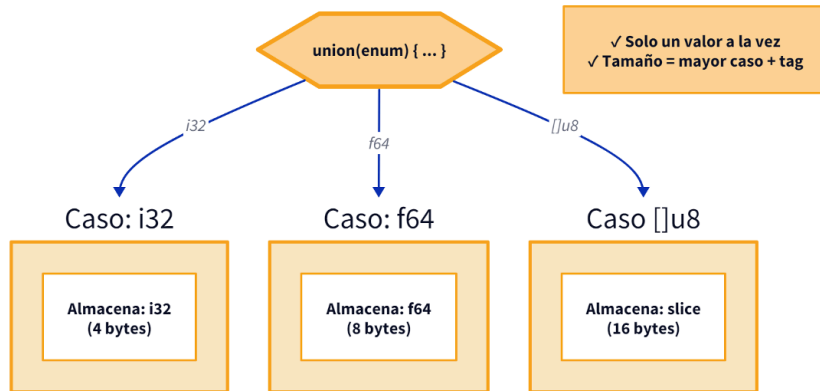


Figura 13 - Unión etiquetada

Las uniones etiquetadas son muy útiles para representar datos que pueden tener diferentes estructuras. Por ejemplo, un mensaje en una aplicación de chat podría ser texto, una imagen, o un archivo:

1	<code>const Mensaje = unión(enum) {</code>
2	<code> texto: []const u8,</code>
3	<code> imagen: struct {</code>
4	<code> url: []const u8,</code>
5	<code> ancho: u32,</code>
6	<code> alto: u32,</code>
7	<code> },</code>
8	<code> archivo: struct {</code>
9	<code> nombre: []const u8,</code>
10	<code> tamaño: u64,</code>
11	<code> },</code>
12	<code>};</code>



Consejo

Las uniones etiquetadas son la forma idiomática en Zig de representar tipos que pueden tener diferentes formas. Son similares a los 'algebraic data types' de lenguajes funcionales o a los 'sealed classes' de otros lenguajes orientados a objetos.

Aspecto	Struct	Enum	Unión
Contiene	Todos los campos	Una variante	Una variante + dato
Tamaño	Suma de campos	Mínimo necesario	Mayor variante
Acceso	Notación punto	Comparación	Switch

Tabla 39 – Comparación de Tipos Personalizados
(Fuente: Propia)

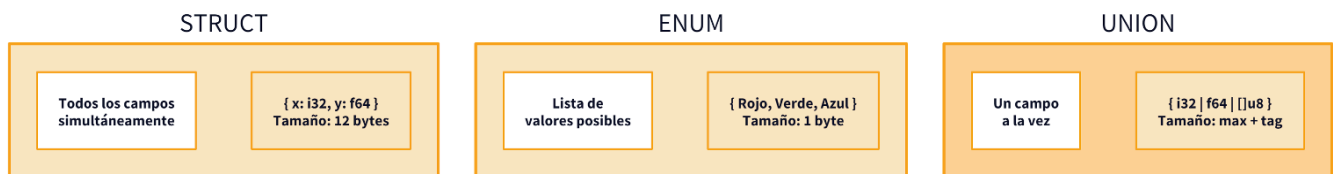


Figura 14 - Struct vs enum vs unión

A veces necesitas una estructura para un uso único y no vale la pena darle un nombre. Zig permite estructuras anónimas usando la sintaxis `.{}`:

1	<code>fn crear_punto(x: f32, y: f32) struct { x: f32, y: f32 } {</code>
2	<code>return .{ .x = x, .y = y };</code>

3	}
4	
5	const p = crear_punto(3.0, 4.0);
6	std.debug.print("{d}, {d}}\n", .{p.x, p.y});

Las estructuras anónimas también se usan en la sintaxis `.{}` que has visto para pasar argumentos a funciones como `std.debug.print`. Los valores entre `{}` son en realidad campos de una estructura anónima.

Las estructuras pueden contener otras estructuras, permitiéndote construir tipos complejos a partir de tipos más simples. Esta técnica se llama composición:

1	const Dirección = struct {
2	calle: []const u8,
3	ciudad: []const u8,
4	código_postal: []const u8,
5	};
6	
7	const Persona = struct {
8	nombre: []const u8,
9	edad: u8,
10	dirección: Dirección,
11	};
12	
13	const juan = Persona{
14	.nombre = "Juan Garcia",
15	.edad = 30,
16	.dirección = Dirección{
17	.calle = "Av. Principal 123",
18	.ciudad = "Madrid",

19	<code>.código_postal = "28001",</code>
20	<code>},</code>
21	<code>};</code>

Para acceder a campos anidados, encadenas la notación de punto. Por ejemplo, `juan.dirección.ciudad` te da "Madrid". Esta capacidad de anidar tipos te permite modelar relaciones complejas de forma natural.

Veamos un programa completo que usa estructuras, enums y métodos para modelar un sistema simple de inventario de productos:

1	<code>const std = @import("std");</code>
2	
3	<code>const Categoría = enum {</code>
4	<code> electronica,</code>
5	<code> ropa,</code>
6	<code> alimentos,</code>
7	<code> libros,</code>
8	
9	<code>pub fn nombre(self: Categoría) []const u8 {</code>
10	<code> dirección: Dirección, return switch (self) {</code>
11	<code> .electronica ⇒ "Electronica",</code>
12	<code> .ropa ⇒ "Ropa",</code>
13	<code> .alimentos ⇒ "Alimentos",</code>
14	<code> .libros ⇒ "Libros",</code>
15	<code> };</code>
16	<code>}</code>
17	<code>};</code>
18	
19	<code>const Producto = struct {</code>
20	<code> nombre: []const u8,</code>

21	precio: f32,
22	cantidad: u32,
23	categoría: Categoría,
24	
25	pub fn valor_inventario(self: Producto) f32 {
26	return self.precio * @as(f32, @floatFromInt(self.cantidad));
27	}
28	
29	pub fn esta_agotado(self: Producto) bool {
30	return self.cantidad == 0;
31	}
32	
33	pub fn mostrar(self: Producto) void {
34	std.debug.print("{s} ({s})\n", .{self.nombre, self.categoría.nombre});
35	std.debug.print(" Precio: \${d:.2}\n", .{self.precio});
36	std.debug.print(" Stock: {}\n", .{self.cantidad});
37	std.debug.print(" Valor total: \${d:.2}\n", . {self.valor_inventario()});
38	}
39	};
40	
41	pub fn main() void {
42	const productos = [_]Producto{
43	.{ .nombre = "Laptop", .precio = 999.99, .cantidad = 5, .categoría = .electronica },
44	.{ .nombre = "Camiseta", .precio = 29.99, .cantidad = 50, .categoría = .ropa },
45	.{ .nombre = "Cafe", .precio = 12.50, .cantidad = 100, .categoría = .alimentos },
46	};

47	
48	<code>var total: f32 = 0;</code>
49	<code>for (productos) prod {</code>
50	<code>prod.mostrar();</code>
51	<code>std.debug.print("\n", .{});</code>
52	<code>total += prod.valor_inventario();</code>
53	<code>}</code>
54	
55	<code>std.debug.print("Valor total del inventario: \${d:.2}\n", .{total});</code>
56	<code>}</code>

```

1  pub fn main() void {
2      const productos = [_]Producto{
3          .{ .nombre = "Laptop", .precio = 999.99, .cantidad = 5, .c
4          .{ .nombre = "Camiseta", .precio = 29.99, .cantidad = 50,
5          .{ .nombre = "Cafe", .precio = 12.50, .cantidad = 100, .ca
6      };
7  }

```

Los métodos de cálculo como `valor_inventario` evitan duplicar lógica.

Las estructuras en Zig pueden usarse para implementar patrones de diseño comunes de la programación orientada a objetos, aunque de una manera más explícita y sin la complejidad de herencia y polimorfismo tradicional.

Patrón	Descripción	Implementación en Zig
Builder	Construcción paso a paso	Métodos encadenados con *self
Factory	Crear objetos variantes	Función que retorna struct
Singleton	Instancia única	Constante global
Strategy	Algoritmos intercambiables	Campos de función

Tabla 40 - Patrones de Diseño con Structs
(Fuente: Propia)

El patrón Builder es útil cuando tienes muchos campos opcionales. En lugar de un constructor con muchos parámetros, ofreces métodos encadenados:

1	<pre>const Configuracion = struct {\n puerto: u16 = 8080,\n host: []const u8 = \"localhost\", \n timeout: u32 = 30,\n ssl: bool = false,\n\n pub fn init() Configuracion {\n return .{}; // Valores por defecto\n }\n\n pub fn conPuerto(self: *Configuracion, p: u16) *Configuracion {\n self.puerto = p;\n return self;\n }\n\n pub fn conSSL(self: *Configuracion) *Configuracion {\n self.ssl = true;\n return self;\n }\n};\n\nvar config = Configuracion.init();\n_ = config.conPuerto(443).conSSL();</pre>
---	--

El patrón Factory crea diferentes variantes de un tipo basándose en parámetros:

1	<pre>const Forma = struct {\n tipo: enum { circulo, rectángulo, triangulo },\n area: f32,\n\n pub fn circulo(radius: f32) Forma {\n return .{ .tipo = .circulo, .area = 3.14159 * radius * radius };\n }\n\n pub fn rectángulo(ancho: f32, alto: f32) Forma {\n return .{ .tipo = .rectángulo, .area = ancho * alto };\n }\n\n pub fn triangulo(base: f32, altura: f32) Forma {\n return .{ .tipo = .triangulo, .area = base * altura / 2 };\n }\n};\n\nconst c = Forma.circulo(5); // Area: 78.54\nconst r = Forma.rectángulo(4, 6); // Area: 24</pre>
---	--

Una de las fortalezas de Zig es su integración nativa con C. Las estructuras pueden declararse como 'extern' para garantizar compatibilidad binaria con bibliotecas C:

Tipo	Layout	Uso
struct	Zig decide	Uso interno
extern struct	Compatible con C	FFI con C

packed struct	Sin padding	Protocolos binarios
---------------	-------------	---------------------

Tabla 41 - Tipos de Struct para Interop
(Fuente: Propia)

1	<pre>// Estructura compatible con C (mismo layout en memoria)\nconst Point = extern struct {\n x: c_int, // Tipos C para compatibilidad\n y: c_int,\n};\n\n// Importar función de C que usa esta estructura\nextern fn c_draw_point(p: Point) void;\n\n// Llamar desde Zig\nconst p = Point{ .x = 100, .y = 200 }; \nc_draw_point(p);</pre>
---	--

Las estructuras packed son útiles para trabajar con formatos binarios exactos, como protocolos de red o formatos de archivo:

1	<pre>// Sin padding - exactamente 4 bytes\nconst Header = packed struct {\n version: u4, // 4 bits\n flags: u4, // 4 bits\n length: u16, // 16 bits\n checksum: u8, // 8 bits\n};\n\n// Sizeof = 4 bytes exactly\nconst size = @sizeof(Header); // 4</pre>
---	---

Advertencia

Las estructuras packed tienen restricciones: no puedes tomar direcciones de campos individuales y el rendimiento puede ser menor. Úsalas solo cuando realmente necesites control exacto del layout de memoria.

En este capítulo aprendiste a crear tipos personalizados usando estructuras, enumeraciones y uniones. Las estructuras te permiten agrupar datos relacionados y pueden incluir métodos para operar sobre esos datos. Los valores por defecto hacen la inicialización más conveniente.

Las enumeraciones son ideales para valores que solo pueden ser una de un conjunto fijo de opciones, y Zig te ayuda a manejar todos los casos con switch. Las uniones etiquetadas

combinan la flexibilidad de poder almacenar diferentes tipos con la seguridad de saber siempre que tipo contienen.

La composición de tipos te permite construir estructuras complejas a partir de tipos más simples, creando modelos claros del mundo real. Estas herramientas son fundamentales para escribir programas organizados y mantenibles.

En el próximo capítulo exploraremos el manejo de errores en Zig, un sistema elegante que te ayuda a escribir código robusto que maneja problemas de forma explícita y predecible.

EJERCICIOS

Pon a prueba tus conocimientos con los siguientes ejercicios

ESTRUCTURA CÍRCULO

Define una estructura 'Circulo' con un campo 'radio' de tipo f32. Agrega métodos para calcular el área ($\pi * r^2$) y la circunferencia ($2 * \pi * r$). Usa 3.14159 como π .

Ejercicio

ENUM SEMÁFORO

Crea un enum 'Semaforo' con variantes rojo, amarillo y verde. Agrega un método 'siguiente' que retorne el siguiente color en la secuencia (verde → amarillo → rojo → verde).

Ejercicio

ESTRUCTURA CUENTA BANCARIA

Crea una estructura 'CuentaBancaria' con campos 'titular' (string), 'saldo' (f64), y métodos 'depositar', 'retirar' (solo si hay saldo suficiente), y 'mostrar_saldo'.

Ejercicio

3

UNIÓN PARA RESULTADO

Crea una unión etiquetada 'Resultado' que pueda ser 'éxito' con un valor i32, 'error' con un mensaje string, o 'pendiente' sin valor.

Ejercicio

4

SISTEMAS DE TAREAS

Crea estructuras para un sistema de tareas: un enum 'Prioridad' (alta, media, baja), un enum 'Estado' (pendiente, en_progreso, completada), y una estructura 'Tarea' con titulo, prioridad, estado, y un método para completar la tarea..

Ejercicio

5

CAPÍTULO 10

Manejo de Errores

Los programas del mundo real enfrentan condiciones imprevistas constantemente: archivos que no existen, conexiones de red que fallan, datos de entrada inválidos, memoria insuficiente. Lo que distingue a un programa robusto de uno frágil es como maneja estas situaciones excepcionales. Muchos lenguajes usan excepciones, que interrumpen el flujo normal del programa cuando algo sale mal. Otros lenguajes más antiguos simplemente retornan códigos de error que fácil mente se ignoran. Zig toma un camino diferente que combina lo mejor de ambos mundos.

El sistema de manejo de errores de Zig está integrado en el sistema de tipos del lenguaje. Los errores son valores de primera clase que deben ser manejados explícitamente; no puedes ignorarlos accidentalmente. Al mismo tiempo, la sintaxis es ligera y no oscurece el flujo principal del programa. Esta combinación te permite escribir código que es simultáneamente conciso y robusto.

En este capítulo aprenderás cómo definir y retornar errores, cómo manejarlos con `try` y `catch`, y cómo asegurar que los recursos se limpien correctamente incluso cuando ocurren errores. Estas técnicas son fundamentales para escribir programas confiables en Zig.

Palabra clave	Función	Uso
<code>try</code>	Propagar error	<code>const x = try func();</code>
<code>catch</code>	Manejar error	<code>func() catch valor;</code>
<code>errdefer</code>	Limpieza si hay error	<code>errdefer liberar();</code>

Tabla 42 - Palabras Clave para Errores
(Fuente: Propia)

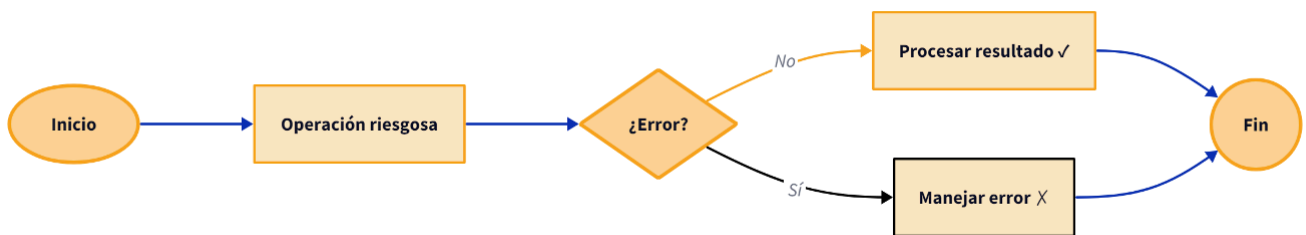


Figura 15 - Flujo de errores

En Zig, los errores no son excepciones ni números mágicos; son valores de un tipo especial llamado error set. Un error set es similar a una enumeración: define un conjunto de posibles errores con nombres descriptivos. Puedes definir tus propios error sets para describir exactamente que puede fallar en tu código:

1	<code>const ArchivoError = error{</code>
2	<code> ArchivoNoEncontrado,</code>
3	<code> PermisosDenegados,</code>
4	<code> DiscoLleno,</code>
5	<code>};</code>

Este error set define tres posibles errores relacionados con operaciones de archivos. Los nombres son descriptivos y autoexplicativos, lo que ayuda a entender que salió mal cuando ocurre un error. Si quieres referirte a un error específico, usas la sintaxis `error.NombreDelError`.

También puedes usar el error set global, que agrupa todos los errores posibles en tu programa. Cuando no especificas un error set particular, Zig usa `anyerror`, que puede contener cualquier error. Esto es conveniente pero menos preciso que definir exactamente que errores puede producir una función.

Las funciones que pueden fallar retornan un tipo especial llamado error unión. Un error unión combina un tipo de error con el tipo de resultado exitoso. La sintaxis usa el símbolo de exclamación: `!T` significa "puede retornar un error o un valor de tipo T":

1	fn dividir(a: f64, b: f64) !f64 {
2	if (b == 0) {
3	} return error.DivisiónPorCero;
4	}
5	return a / b;
6	}

Esta función retorna !f64, que significa "o un error, o un f64". Si b es cero, retorna el error DivisiónPorCero. De lo contrario, retorna el resultado de la división. El llamador de esta función debe decidir **como** manejar el posible error.

Sintaxis	Significado	Ejemplo
!T	Cualquier error o T	!f64
E!T	Error de E o T	MiError!i32
!void	Error o nada	!void

Tabla 43 - Sintaxis de Error Unions
(Fuente: Propia)

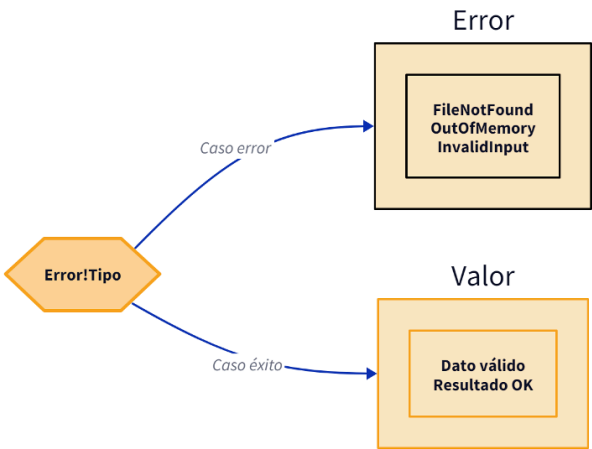


Figura 16 - Error unions

También puedes especificar exactamente que errores puede producir una función colocando el error set antes del signo de exclamación:

1	<code>fn const MiError = error{ ValorInvalido, FueraDeRango };</code>
2	
3	<code>fn procesar(valor: i32) MiError!i32 {</code>
4	<code> if (valor < 0) return error.ValorInvalido;</code>
5	<code> if (valor > 100) return error.FueraDeRango;</code>
6	<code> return valor * 2;</code>
7	<code>}</code>



NOTA

Especificar el error set exacto es una buena práctica porque documenta exactamente que puede fallar, y el compilador puede verificar que el código que llama a la función maneje todos los casos posibles.

Cuando llamas a una función que puede fallar, necesitas decidir qué hacer con el posible error. La forma más simple es usar la palabra clave 'try', que propaga el error al llamador si ocurre:

1	<code>fn calcular() !f64 {</code>
2	<code> const x = try dividir(10, 2); // Si falla, propaga el error</code>
3	<code> const y = try dividir(x, 3); // Si falla, propaga el error</code>
4	<code> return x + y;</code>

5	}
---	---

Si dividir retorna un error, try inmediatamente retorna ese mismo error desde calcular. Si dividir tiene éxito, try desenvuelve el valor y lo asigna a la variable. Esto crea un flujo de control muy limpio donde los errores se propagan automáticamente hacia arriba en la pila de llamadas.

La función calcular también retorna un error unión (!f64) porque usa try internamente. Si una función usa try, debe ser capaz de retornar errores. Esto es intencional: Zig te obliga a ser explícito sobre que funciones pueden fallar.

A veces quieres manejar el error localmente en lugar de propagarlo. Para esto existe catch, que te permite especificar qué hacer cuando ocurre un error:

1	fn dividir_seguro(a: f64, b: f64) f64 {
2	const resultado = dividir(a, b) catch err {
3	std.debug.print("Error: {}\n", .{err});
4	return 0; // Valor por defecto si hay error
5	};
6	return resultado;
7	}

El bloque catch captura el error (disponible como |err|) y te permite decidir qué hacer. En este caso, imprimimos el error y retornamos un valor por defecto. Observa que dividir_seguro retorna f64 (sin !), no un error unión, porque todos los posibles errores son manejados internamente.

Si solo quieres proporcionar un valor por defecto sin manejar el error de forma especial, puedes usar la forma abreviada de catch:

1	const resultado = dividir(10, 0) catch 0; // Si falla, usa 0
---	--

También puedes usar catch con diferentes errores de forma diferente usando un switch implícito:

1	const resultado = leer_archivo() catch err switch (err) {
2	error.ArchivoNoEncontrado => {
3	std.debug.print("Archivo no existe, creando uno nuevo\n", .{});

4	<code>return crear_archivo();</code>
5	<code>},</code>
6	<code>error.PermisosDenegados => {</code>
7	<code>std.debug.print("No tienes permiso\n", .{});</code>
8	<code>return error.PermisosDenegados; // Re-propagar este error</code>
9	<code>},</code>
10	<code>else => return err, // Propagar cualquier otro error</code>
11	<code>};</code>

Finalmente, existe una forma muy elegante de manejar errores cuando quieres procesar tanto el caso de éxito como el de fallo en bloques separados: usar 'if' con captura de error. Esto es similar a como funcionan los opcionales:

1	<code>if (operacion()) valor </code>
2	<code>std.debug.print("Éxito: {}\n", .{valor});</code>
3	<code>} else err {</code>
4	<code>std.debug.print("Fallo: {s}\n", .{@errorName(err)});</code>
5	<code>}</code>

Ya conoces defer, que ejecuta código de limpieza cuando el bloque actual termina. Zig también tiene errdefer, que solo ejecuta el código si la función retorna un error. Esto es extremadamente útil para limpiar recursos cuando algo sale mal:

1	<code>fn crear_conexion() !*Conexion {</code>
2	<code>const mem = try allocator.alloc(u8, 1024);</code>
3	<code>errdefer allocator.free(mem); // Liberar si algo falla después</code>
4	
5	<code>const socket = try abrir_socket();</code>
6	<code>errdefer cerrar_socket(socket); // Cerrar si algo falla después</code>
7	
8	<code>try configurar_conexion(socket);</code>
9	

10	<code>return conexion;</code>
11	<code>}</code>

Si `configurar_conexion` falla, ambos `errdefers` se ejecutan: se cierra el socket y se libera la memoria. Si todo tiene éxito, los `errdefers` no se ejecutan porque no hubo error. Esto garantiza que no haya fugas de recursos cuando algo sale mal a mitad de una operación compleja.

La combinación de `try` y `errdefer` crea un patrón muy poderoso para código que necesita adquirir múltiples recursos. Cada recurso se protege con un `errdefer` inmediatamente después de adquirirlo, asegurando limpieza automática si cualquier paso posterior falla



Consejo

Cuando adquieras recursos (memoria, archivos, conexiones), sigue el patrón: (1) adquirir recurso, (2) inmediatamente después, `errdefer` para liberarlo, (3) continuar con operaciones que pueden fallar. Esto hace prácticamente imposible tener fugas de recursos.

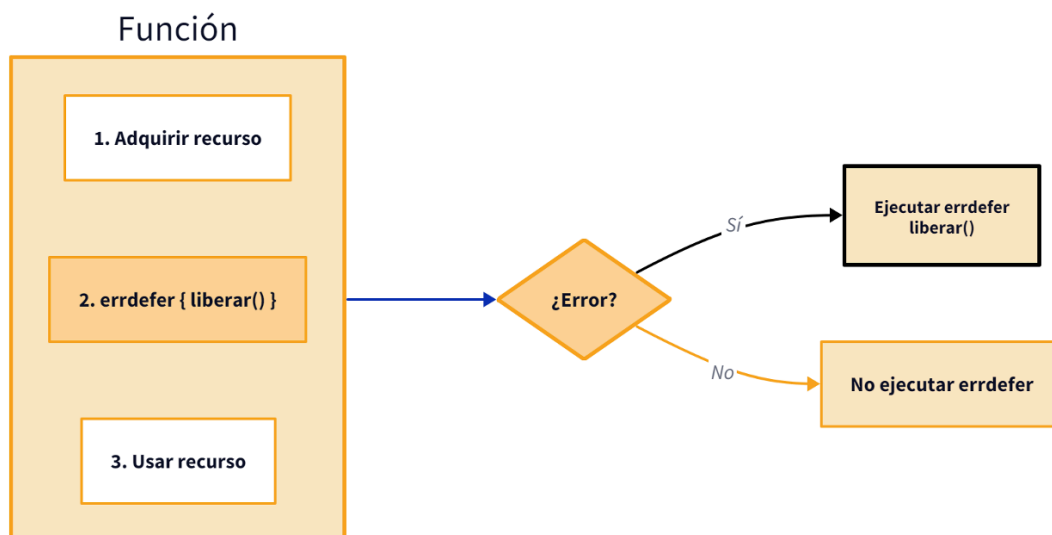


Figura 17 - Errdefer

A veces estas absolutamente seguro de que una operación no puede fallar en una situación particular. En estos casos, puedes usar `catch unreachable` para indicar que un error sería un bug en tu programa:

1	<code>// Sabemos que esto no puede fallar porque ya validamos el input</code>
2	<code>const número = std.fmt.parseInt(i32, "42", 10) catch unreachable;</code>

Si la operación falla y llegas a `unreachable`, el programa termina con un `panic`. Usa esto con cuidado y solo cuando realmente estes seguro de que el error es imposible dado el contexto.

También puedes causar un `panic` explícitamente con `@panic` cuando encuentras una condición que no debería ser posible:

1	<code>fn procesar_comando(cmd: Comando) void {</code>
2	<code> switch (cmd) {</code>
3	<code> .iniciar => iniciar(),</code>
4	<code> .detener => detener(),</code>
5	<code> .reiniciar => {</code>
6	<code> detener();</code>
7	<code> iniciar();</code>
8	<code> },</code>
9	<code> _ => @panic("Comando no implementado"),</code>
10	<code> }</code>
11	<code>}</code>



NOTA

Un panic no es lo mismo que un error manejable. Los errores son para condiciones que pueden ocurrir normalmente (archivo no existe, red no disponible). Los panics son para bugs o condiciones que no deberían ocurrir nunca. Un panic siempre termina el programa.

La función `main` puede retornar un error unión, lo que permite que los errores se propaguen hasta el nivel más alto del programa. Si `main` retorna un error, Zig imprime información sobre el error y termina con un código de salida no cero:

1	<code>pub fn main() !void {</code>
2	<code> const archivo = try abrir_archivo("datos.txt");</code>
3	<code> defer cerrar_archivo(archivo);</code>
4	<code> const contenido = try leer_todo(archivo);</code>
5	<code> try procesar(contenido);</code>
6	<code> std.debug.print("Procesamiento completado\n", .{});</code>
7	<code>}</code>

Este patrón es muy común para programas que realizan operaciones de E/S o cualquier cosa que pueda fallar. Los errores fluyen naturalmente hacia `main`, donde Zig se encarga de mostrar información útil sobre que salió mal.

Para crear tus propios errores, simplemente defines un error set y usas sus valores. Los nombres deben ser descriptivos y seguir la convención `PascalCase`:

1	<code>const ValidaciónError = error{</code>
2	<code> EmailInvalido,</code>

3	ContrasenaDebil,
4	UsuarioExiste,
5	EdadInsuficiente,
6	};
7	
8	fn validar_registro({
9	email: []const u8,
10	password: []const u8,
11	edad: u8,
12	) ValidaciónError!void {
13	if (!contiene(email, '@')) return error.EmailInvalido;
14	if (password.len < 8) return error.ContrasenaDebil;
15	if (edad < 18) return error.EdadInsuficiente;
16	// Todo valido, no retorna error
17	}

Observa que la función retorna ValidaciónError!void, lo que significa que si tiene éxito no retorna ningún valor (void), pero puede fallar con uno de los errores de ValidaciónError.

Patrón	Sintaxis	Resultado
Valor por defecto	func() catch 0	0 si hay error
Convertir a opcional	func() catch null	?T
Log y propagar	func() catch err {...; return err;}	Log + error

Tabla 44 - Patrones Comunes de Manejo de Errores
(Fuente: Propia)

Hay varios patrones comunes para el manejo de errores en Zig. Uno es el patrón de conversión a opcional, donde conviertes un error unión a un opcional si solo te importa el valor exitoso:

1	<code>const valor: ?i32 = operacion_falible() catch null;</code>
---	--

Otro patrón es proporcionar una función alternativa de fallback que no puede fallar:

1	<code>fn obtener_config() Config {</code>
2	<code> return cargar_desde_archivo() catch obtener_config_default();</code>
3	<code>}</code>

También es común usar `@errorName` para obtener el nombre de un error como string, útil para logging y mensajes de error:

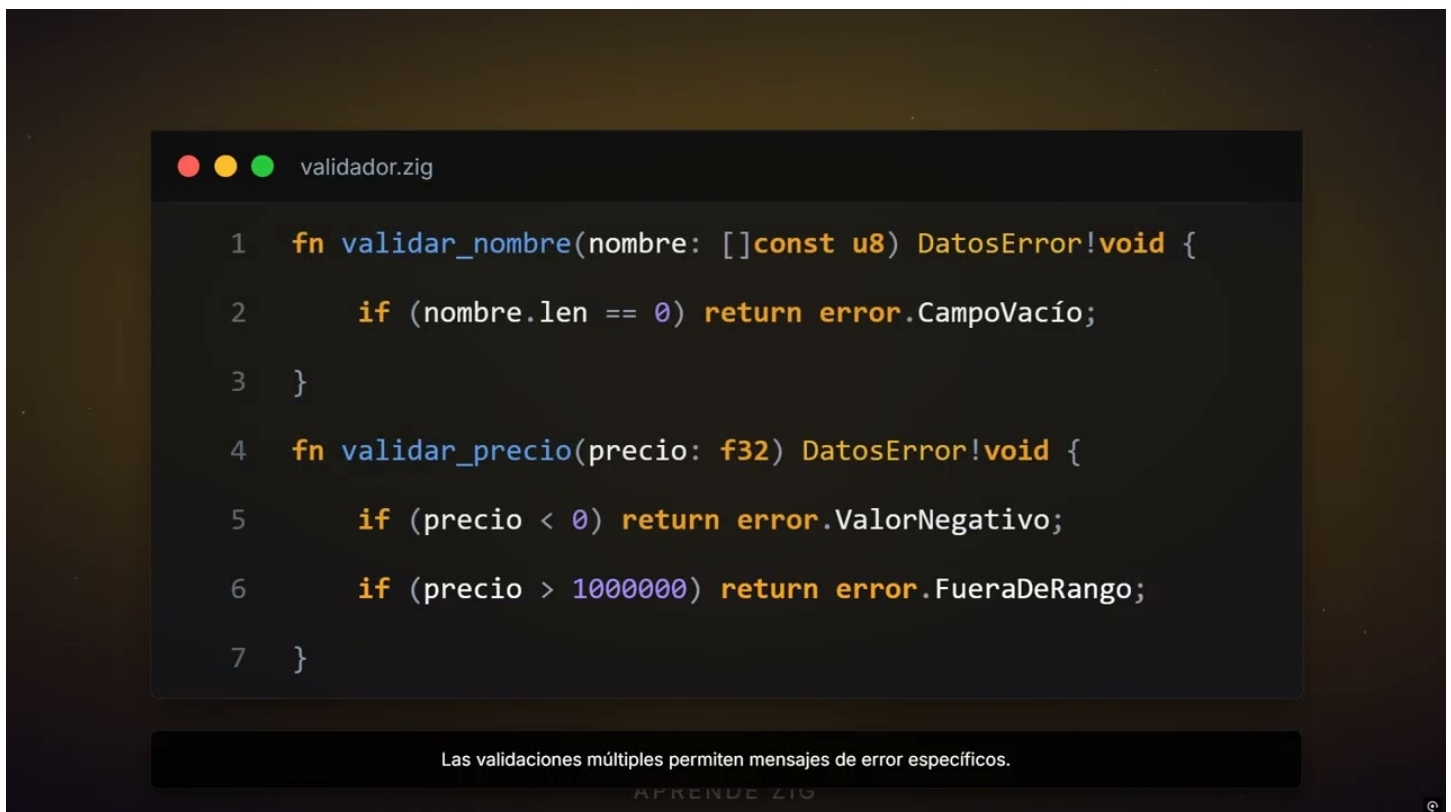
1	<code>const resultado = operacion() catch err {</code>
2	<code> std.debug.print("Error: {s}\n", .{@errorName(err)});</code>
3	<code> return err;</code>
4	<code>};</code>

Veamos un programa completo que implementa un validador de datos con manejo de errores robusto:

1	<code>const std = @import("std");</code>
2	
3	<code>const DatosError = error{</code>
4	<code> CampoVacío,</code>
5	<code> ValorNegativo,</code>
6	<code> FormatoInvalido,</code>
7	<code> FueraDeRango,</code>
8	<code>};</code>
9	
10	<code>const Producto = struct {</code>
11	<code> nombre: []const u8,</code>

12	precio: f32,
13	cantidad: u32,
14	};
15	
16	fn validar_nombre(nombre: []const u8) DatosError!void {
17	if (nombre.len == 0) return error.CampoVacío;
18	}
19	
20	fn validar_precio(precio: f32) DatosError!void {
21	if (precio < 0) return error.ValorNegativo;
22	if (precio > 1000000) return error.FueraDeRango;
23	}
24	
25	fn validar_producto(p: Producto) DatosError!Producto {
26	try validar_nombre(p.nombre);
27	try validar_precio(p.precio);
28	return p;
29	}
30	
31	fn procesar_producto(p: Producto) void {
32	const validado = validar_producto(p) catch err {
33	std.debug.print("Producto invalido: {s}\n", .{@errorName(err)});
34	return;
35	};
36	std.debug.print("Producto valido: {s}, \${d:.2}\n",
37	.{validado.nombre, validado.precio});
38	}
39	

40	pub fn main() void {
41	const productos = [_]Producto{
42	.{ .nombre = "Laptop", .precio = 999.99, .cantidad = 5 },
43	.{ .nombre = "", .precio = 50, .cantidad = 10 }, // Error: vacío
44	.{ .nombre = "Mouse", .precio = -5, .cantidad = 20 }, // Error: negativo
45	};
46	
47	for (productos) p {
48	procesar_producto(p);
49	}
50	}



Aunque el sistema de errores de Zig es robusto, hay algunos patrones problemáticos que debes evitar. Conocer estos errores comunes te ayudará a escribir código más confiable desde el principio.

Error	Problema	Solución
Ignorar errores	catch _ {} vacío	Manejar o propagar el error
Errores genéricos	error.GenericError	Crear error sets específicos
Panic excesivo	unreachable en lugar de error	Usar errores recuperables
Error unión innecesario	Función nunca falla	Retornar valor directo

Tabla 45 - Errores Comunes y Cómo Evitarlos
(Fuente: Propia)

El error más común es ignorar errores con un catch vacío. Esto oculta problemas que pueden manifestarse mucho después, haciendo el debugging muy difícil:

1	<pre>// MAL: Ignora el error silenciosamente\nconst valor =\noperacion_riesgosa() catch _ {} \n // No hacer nada... ¡peligroso!\n}\n\n// BIEN: Manejar explícitamente\nconst valor = operacion_riesgosa()\ncatch err {\n std.log.err(\"Operacion fallo: {s}\", .{\n @errorName(err)});\n return error.OperationFallida;\n};</pre>
---	---

Otro problema común es usar error sets demasiado genéricos. Errores específicos hacen el código más fácil de entender y depurar:

1	<pre>/// MAL: Error genérico sin contexto\nconst ConfigError =\nerror{ConfigError};\n\n// BIEN: Errores específicos y descriptivos\nconst ConfigError = error{\n ArchivoNoEncontrado,\n FormatoInvalido,\n ValorFueraDeRango,\n PermisosDenegados,\n};, .{@errorName(err)};\n\nreturn error.OperationFallida;\n};</pre>
---	---



Probar que tu código maneja errores correctamente es tan importante como probar los casos exitosos. Zig tiene soporte integrado para tests que facilita verificar el comportamiento de errores.

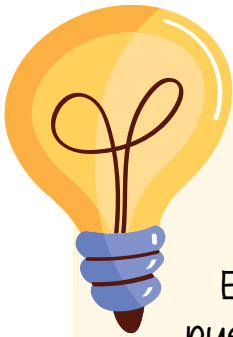
Función	Propósito	Ejemplo
expectError	Verifica error específico	expectError(.DivPorCero, fn())
expectEqual	Verifica valor exitoso	expectEqual(valor, resultado)
expect	Condición general	expect(valor > 0)

Tabla 46 - Funciones de Testing para Errores

Puedes escribir tests que verifiquen tanto los casos exitosos como los de error:

1	<pre>const std = @import(\"std\");\nconst expect = std.testing.expect;\nconst expectError = std.testing.expectError;\n\nfn dividir(a: i32, b: i32) !i32 {\n if (b == 0) return error.DivisiónPorCero;\n return @divTrunc(a, b);\n}\n\ntest \"división exitosa\" {\n const resultado = try dividir(10, 2);\n try expect(resultado == 5);\n}\n\ntest \"división por cero retorna error\" {\n try expectError(error.DivisiónPorCero, dividir(10, 0));\n}</pre>
---	--

Para correr los tests, usa el comando 'zig build test' o 'zig test archivo.zig'. Zig reportará cuáles tests pasaron y cuáles fallaron, con información detallada sobre las fallas.



Consejo

Escribe tests para todos los tipos de errores que tu función puede retornar. Esto documenta el comportamiento esperado y previene regresiones cuando modifiques el código en el futuro.

En este capítulo aprendiste el sistema de manejo de errores de Zig, que está integrado en el sistema de tipos del lenguaje.

La palabra clave `try` propaga errores automáticamente al llamador, manteniendo el código limpio. La palabra clave `catch` te permite manejar errores localmente, ya sea proporcionando un valor por defecto o ejecutando código alternativo.

Viste como `errdefer` garantiza la limpieza de recursos cuando una función falla, complementando a `defer` para crear código robusto que no tiene fugas.

EJERCICIOS

Pon a prueba tus conocimientos con los siguientes ejercicios

DIVISIÓN SEGURA

Crea una función 'dividir' que reciba dos enteros y retorne un error unión. Debe retornar error.DivisiónPorCero si el divisor es cero, de lo contrario el resultado de la división.

Ejercicio

VALIDADOR DE EDAD

Crea un error set con EdadNegativa, MenorDeEdad, y EdadIrreal (mayor a 150). Escribe una función 'validar_edad' que valide un número y lo retorne si es válido.

Ejercicio

PROPAGANDA CON TRY

Crea tres funciones que se llaman entre sí: `paso1`, `paso2`, `paso3`. Cada una puede fallar con un error diferente. La función `paso1` llama a `paso2`, que llama a `paso3`. Usa `try` para propagar errores.

Ejercicio

3

CAPÍTULO II

Comptime y Metaprogramación

Una de las características más distintivas y poderosas de Zig es su capacidad para ejecutar código en tiempo de compilación. Mientras la mayoría de los lenguajes distinguen claramente entre lo que sucede cuando compilas un programa y lo que sucede cuando lo ejecutas, Zig difumina esta línea de una manera elegante y útil. Esta capacidad te permite escribir código genérico, realizar cálculos complejos durante la compilación, y generar código optimizado sin el costo de rendimiento que normalmente tendrían estas operaciones.

El nombre para esta característica es "comptime", una contracción de "compile time" (tiempo de compilación). Cuando marcas algo como comptime, le estas diciendo a Zig que debe evaluar esa expresión, variable o parámetro durante la compilación, no cuando el programa se ejecute. Esto abre un mundo de posibilidades que exploraremos en este capítulo.

La metaprogramación, en términos generales, es escribir código que manipula o genera otro código. En muchos lenguajes esto se logra con macros complejas o preprocesadores. Zig toma un enfoque diferente: usa el mismo lenguaje tanto para el código normal como para la metaprogramación, haciendo que sea más fácil de entender y depurar.

Contexto	Sintaxis	Propósito
Variable	comptime var x = ...	Valor en compilación
Parámetro	fn f(comptime T: type)	Tipo genérico
Bloque	comptime { ... }	Cálculos complejos

Tabla 47 - Usos de comptime
(Fuente: Propia)

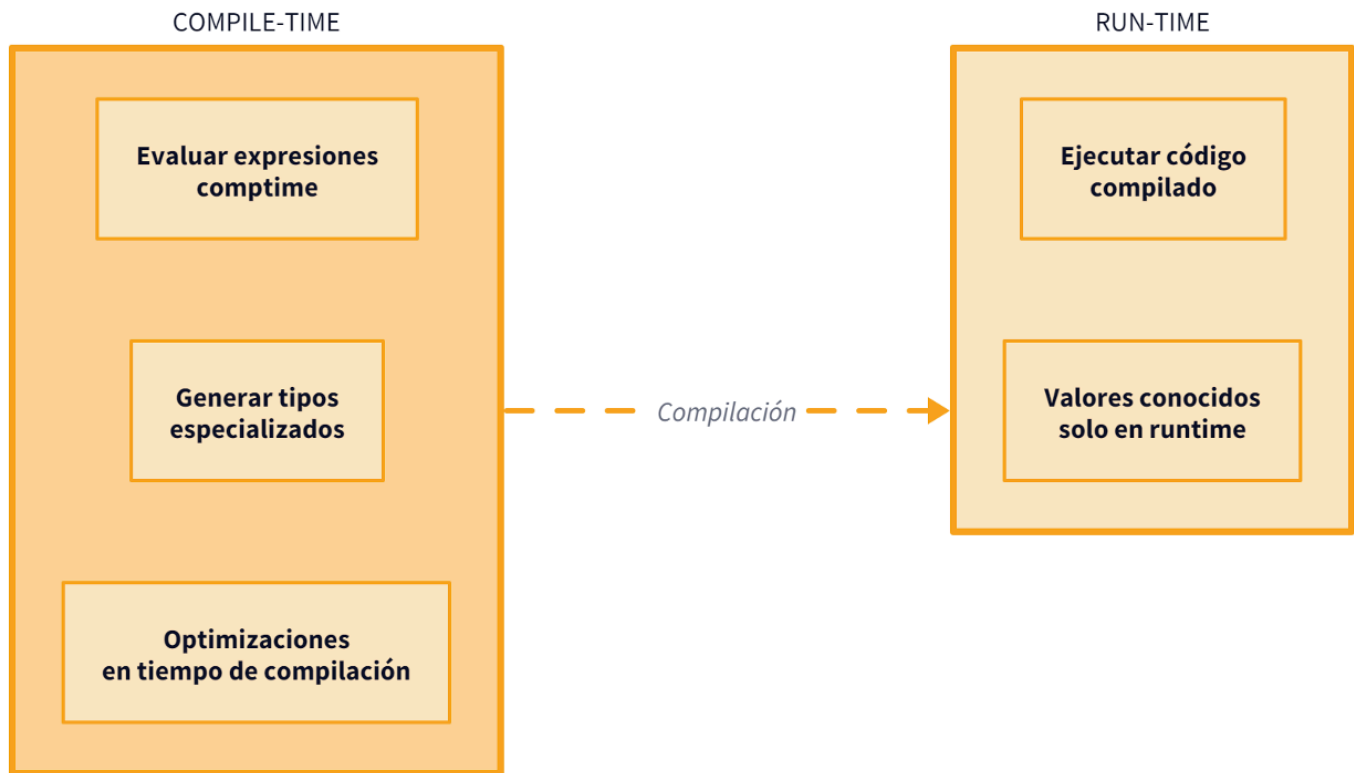


Figura 18 - Comptime overview

Empecemos con lo más básico: los valores comptime. Un valor comptime es un valor que el compilador conoce completamente durante la compilación. Las constantes literales son valores comptime por definición:

1	<code>const x = 5;</code>	<code>// 5 es conocido en tiempo de compilación</code>
2	<code>const y = 3.14;</code>	<code>// 3.14 es conocido en tiempo de compilación</code>
3	<code>const name = "Zig";</code>	<code>// "Zig" es conocido en tiempo de compilación</code>

Cuando usas `const` para declarar una variable con un valor literal, ese valor existe solo durante la compilación; no ocupa espacio en memoria cuando el programa se ejecuta. El compilador reemplaza cada uso de esa constante con el valor real.

Puedes hacer cálculos con valores comptime, y el resultado también será comptime. El compilador realiza estos cálculos por ti:

1	<code>const segundos_por_dia = 60 * 60 * 24; // Calculado en compilación: 86400</code>
2	<code>const pi = 3.14159;</code>
3	<code>const dos_pi = pi * 2; // Calculado en compilación: 6.28318</code>

Esto no es magia; es simplemente el compilador haciendo matemáticas antes de generar el ejecutable. El beneficio es que tu programa ya tiene estos valores precalculados cuando se ejecuta, sin gastar tiempo en calcularlos.

La palabra clave `comptime` te permite forzar que algo sea evaluado en tiempo de compilación. Puedes usarla con variables para crear valores que deben conocerse durante la compilación:

1	<code>fn calcular_tamaño(n: u32) u32 {</code>
2	<code> return n * n;</code>
3	<code>}</code>
4	
5	<code>// Este cálculo DEBE ocurrir en tiempo de compilación</code>
6	<code>const tamaño = comptime calcular_tamaño(10); // 100, calculado al compilar</code>

En este ejemplo, `calcular_tamaño` es una función normal que podría ejecutarse en tiempo de ejecución o de compilación. Al usar `comptime`, le decimos a Zig que debe evaluar la llamada durante la compilación. Si por alguna razón no pudiera (por ejemplo, si el argumento no fuera conocido en compilación), sería un error.

NOTA

Si una función solo tiene sentido en tiempo de compilación, su resultado no necesita guardarse en memoria cuando el programa se ejecuta. El compilador simplemente usa el resultado directamente donde se necesite.

Una de las aplicaciones más útiles de comptime es en los parámetros de funciones. Un parámetro comptime debe ser conocido cuando llamas a la función, no cuando el programa se ejecuta. Esto es la base de los genéricos en Zig:

1	<code>fn multiplicar(comptime T: type, a: T, b: T) T {</code>
2	<code> return a * b;</code>
3	<code>}</code>
4	
5	<code>const resultado_int = multiplicar(i32, 5, 3); // 15</code>
6	<code>const resultado_float = multiplicar(f64, 2.5, 4.0); // 10.0</code>

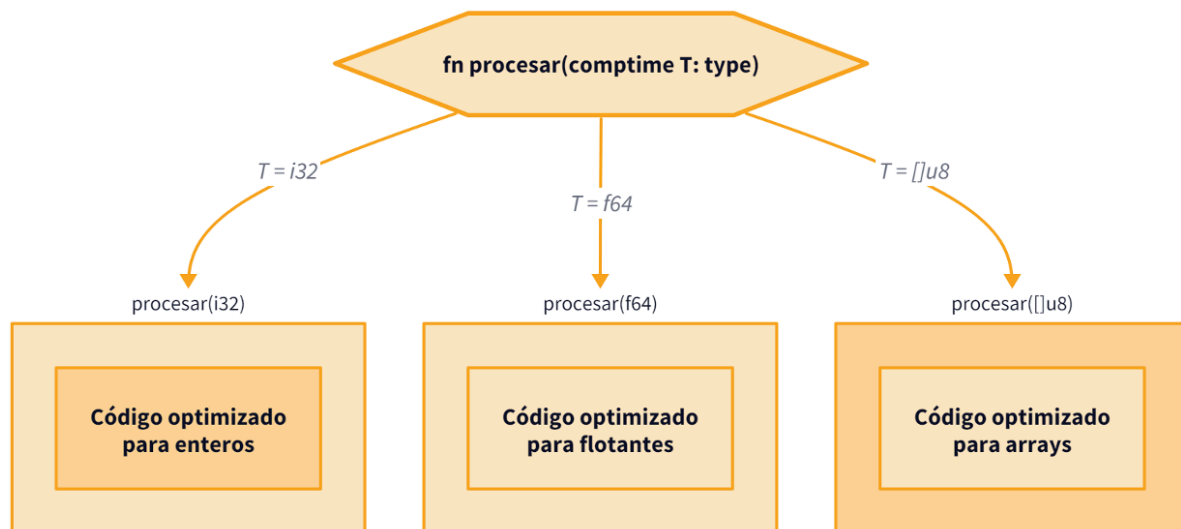


Figura 19 - Especialización genérica

El parámetro T es de tipo "type", que es el tipo de los tipos en Zig. Cuando llamas a multiplicar con diferentes tipos, Zig genera versiones especializadas de la función para cada tipo. Esto es similar a los templates de C++ o los genéricos de otros lenguajes, pero la sintaxis es la misma que para cualquier otra función.

Puedes usar parámetros comptime que no sean tipos también. Por ejemplo, para crear arrays de diferentes tamaños:

1	fn crear_secuencia(comptime n: usize) [n]i32 {
2	var resultado: [n]i32 = undefined;
3	for (0..n) i {
4	resultado[i] = @intCast(i + 1);
5	}
6	return resultado;
7	}
8	
9	const seq_5 = crear_secuencia(5); // [1, 2, 3, 4, 5]
10	const seq_10 = crear_secuencia(10); // [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

Aquí, `n` es conocido en tiempo de compilación, lo que permite que la función retorne arrays de diferentes tamaños. Cada llamada con un `n` diferente genera una versión diferente de la función.

A veces necesitas ejecutar varias líneas de código en tiempo de compilación. Para esto, puedes usar un bloque `comptime`:

1	const tabla_cuadrados = comptime blk: {
2	var resultado: [10]i32 = undefined;
3	for (0..10) i {
4	resultado[i] = @intCast(i * i);
5	}
6	break :blk resultado
7	}; // tabla_cuadrados = [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]

Este código calcula una tabla de cuadrados durante la compilación. Cuando tu programa se ejecuta, la tabla ya está precalculada; no hay bucle, no hay multiplicaciones, solo los valores finales listos para usar.

Características	¿Compile-time?	Ejemplo
Tipos	Si	<code>type MyInt = i32;</code>
<code>const</code>	Si	<code>const x = 5 + 3;</code>
<code>comptime var</code>	Si	<code>comptime var y = 0;</code>
<code>var</code>	No	<code>var z = input();</code>
Funciones inline	Si	<code>inline fn suma() {...}</code>

Tabla 48 - ¿Qué se evalúa en compile-time?
(Fuente: Propia)

Los bucles `for` pueden usar la palabra clave `inline` para "desenrollar" el bucle en tiempo de compilación. Esto significa que el compilador genera código separado para cada iteración:

1	<code>fn imprimir_tipos(comptime tipos: anytype) void {</code>
2	<code> inline for (tipos) T {</code>
3	<code> std.debug.print("Tipo: {s}, tamaño: {} \n",</code>
4	<code> .{@typeName(T), @sizeOf(T)});</code>
5	<code> }</code>
6	<code>}</code>
7	
8	<code>imprimir_tipos({i8, i32, f64, bool});</code>

El `inline for` se expande en tiempo de compilación a cuatro llamadas separadas a `print`, cada una especializada para su tipo. Esto es útil cuando necesitas hacer algo diferente para cada tipo en una colección.



Consejo

Usa inline for cuando necesites iterar sobre una colección de tipos o cuando cada iteración deba generar código diferente. Para iteraciones normales sobre datos, el for regular es más apropiado.

Zig proporciona funciones integradas para examinar tipos en tiempo de compilación. Esto te permite escribir código que se adapta según las características del tipo que recibe:

1	<code>fn describir_tipo(comptime T: type) void {</code>
2	<code> const info = @typeInfo(T);</code>
3	
4	<code> std.debug.print("Tipo: {s}\n", .{@typeName(T)});</code>
5	<code> std.debug.print("Tamaño: {} bytes\n", .{@sizeof(T)});</code>
6	
7	<code> switch (info) {</code>
8	<code> .int => int_info {</code>
9	<code> std.debug.print("Es entero de {} bits\n", .{int_info.bits});</code>
10	<code> },</code>
11	<code> .float => float_info {</code>
12	<code> std.debug.print("Es flotante de {} bits\n", .</code> <code>{float_info.bits});</code>
13	<code> },</code>
14	<code> .@"struct" => std.debug.print("Es una estructura\n", .{}),</code>
15	<code> else => std.debug.print("Otro tipo\n", .{}),</code>
16	<code> }</code>
17	<code>}</code>

La función `@typeInfo` retorna información detallada sobre cualquier tipo. Con esta información puedes tomar decisiones sobre como procesar datos de ese tipo. Esto es extremadamente útil para escribir código genérico que funcione con muchos tipos diferentes.

Otras funciones útiles para reflexión incluyen `@TypeOf` para obtener el tipo de una expresión, `@typeName` para obtener el nombre como string, y `@sizeof` para obtener el tamaño en bytes:

1	<code>const x: i32 = 42;</code>
2	<code>const Tipo = @TypeOf(x); // i32</code>
3	<code>const nombre = @typeName(Tipo); // "i32"</code>
4	<code>const bytes = @sizeof(Tipo); // 4</code>

Función	Reforma	Uso
<code>@typeInfo(T)</code>	Información del tipo	Examinar estructura
<code>@TypeOf(expr)</code>	Tipo de expresión	Inferir tipo
<code>@typeName(T)</code>	Nombre como string	Debugging
<code>@sizeof(T)</code>	Tamaño en bytes	Cálculos de memoria

Tabla 49 - Funciones de Reflexion
(Fuente: Propia)

En Zig, los tipos son valores de primera clase en tiempo de compilación. Puedes almacenarlos en variables, pasarlos como parámetros, y retornarlos de funciones. Esto permite construir tipos dinámicamente:

1	<code>fn Array(comptime T: type, comptime size: usize) type {</code>
2	<code> return struct {</code>
3	<code> data: [size]T,</code>
4	<code> len: usize,</code>
5	
6	<code> pub fn init() @This() {</code>
7	<code> return .{ .data = undefined, .len = 0 };</code>

8	}
9	
10	pub fn append(self: *@This(), value: T) void {
11	self.data[self.len] = value;
12	self.len += 1;
13	}
14	};
15	}
16	
17	var arr = Array(i32, 10).init();
18	arr.append(42);
19	arr.append(100);

Esta función Array toma un tipo y un tamaño, y retorna un nuevo tipo: una estructura con un array interno y métodos. Cada combinación de T y size produce un tipo diferente. Array(i32, 10) es un tipo distinto de Array(f64, 5).

NOTA

@This() es una función integrada que retorna el tipo de la estructura actual. Es útil cuando defines métodos dentro de una estructura anónima generada por una función.

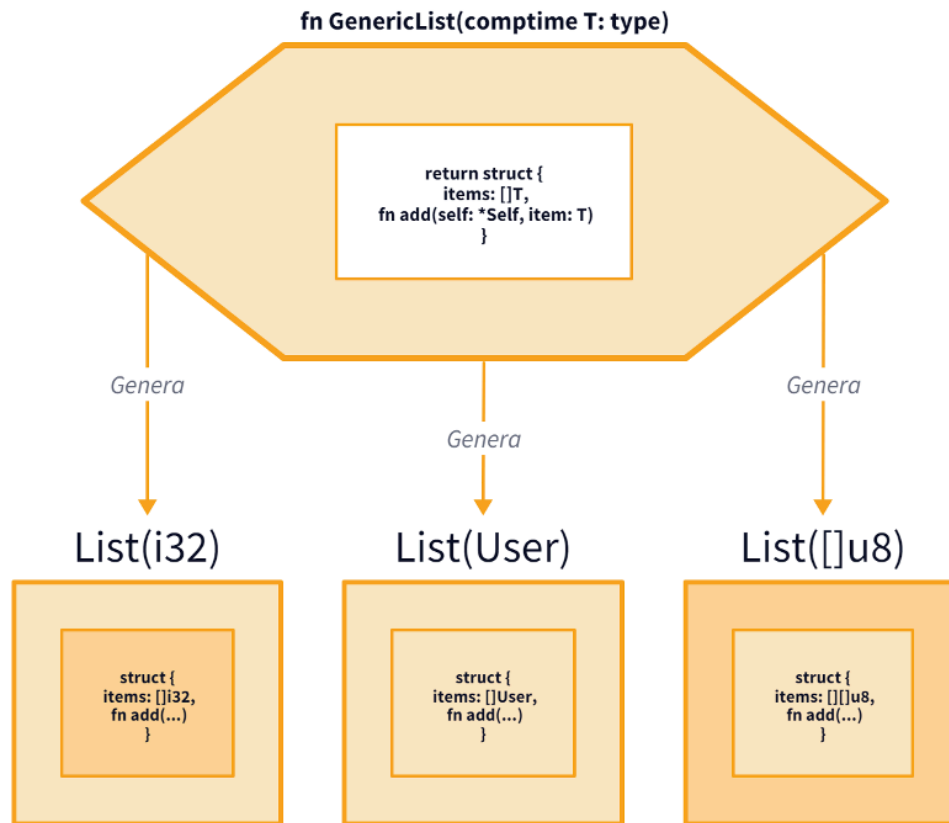


Figura 20 - Type factory

Puedes usar `if` con `comptime` para incluir o excluir código basándote en condiciones conocidas en tiempo de compilación. El código de las ramas no tomadas ni siquiera se compila:

1	<code>fn max_valor(comptime T: type) T {</code>
2	<code> const info = @typeInfo(T);</code>
3	<code> if (info == .int) {</code>
4	<code> if (info.int.signedness == .signed) {</code>
5	<code> return @as(T, (1 << (info.int.bits - 1)) - 1);</code>
6	<code> } else {</code>
7	<code> return @as(T, @intCast(@as(u128, 1 << info.int.bits) - 1));</code>
8	<code> }</code>
9	<code> }</code>
10	<code>@compileError("Solo tipos enteros soportados");</code>

11	}
----	---

La función `@compileError` te permite generar un error de compilación con un mensaje personalizado. Esto es útil para validar que los tipos o valores proporcionados cumplan ciertas condiciones.

Puedes iterar sobre los campos de una estructura en tiempo de compilación, lo que permite escribir funciones que trabajan con cualquier estructura:

1	<code>fn imprimir_campos(comptime T: type, valor: T) void {</code>
2	<code> const fields = @typeInfo(T).@"struct".fields;</code>
3	<code> inline for (fields) field {</code>
4	<code> std.debug.print("{s} = {}\\n", .{field.name, @field(valor, field.name)});</code>
5	<code> }</code>
6	<code>}</code>
7	
8	<code>const Punto = struct { x: i32, y: i32 };</code>
9	<code>imprimir_campos(Punto, .{ .x = 10, .y = 20 });</code>

Este patrón es increíblemente útil para serialización, debugging, clonación de estructuras, y muchas otras tareas que normalmente requerirían escribir código repetitivo para cada tipo de estructura.

Las capacidades `comptime` de Zig tienen muchas aplicaciones prácticas. Una común es crear tablas de búsqueda precalculadas para operaciones que serían costosas en tiempo de ejecución:

1	<code>const tabla_seno = comptime blk: {</code>
2	<code> var t: [360]f32 = undefined;</code>
3	<code> for (0..360) i {</code>
4	<code> const radianes = @as(f32, @floatFromInt(i)) * 3.14159 / 180.0;</code>
5	<code> t[i] = @sin(radianes);</code>
6	<code> }</code>

7	break :blk t;
8	};

Otra aplicación es la validación en tiempo de compilación. Puedes verificar que los datos o configuraciones sean correctos antes de que el programa siquiera exista:

Veamos un programa completo que aprovecha comptime para crear un sistema de serialización simple que funciona con cualquier estructura:

1	const std = @import("std");
2	
3	fn Serializable(comptime T: type) type {
4	{
5	valor: T,
6	
7	pub fn describir(self: @This()) void {
8	const info = @typeInfo(T);
9	if (info != @"struct") {
10	std.debug.print("Tipo: {s} = {}\n", .{@typeName(T), self.valor});
11	return;
12	}
13	
14	std.debug.print("{s}({}\n", .{@typeName(T)});
15	inline for (info.@"struct".fields) field {
16	std.debug.print(" {s} = {},\n",
17	.{field.name, @field(self.valor, field.name)});
18	}
19	std.debug.print(")\n", .{ });
20	}
21	};

22	}
23	
24	const Persona = struct {
25	edad: u8,
26	altura: u16,
27	};
28	
29	pub fn main() void {
30	const p = Serializable(Persona){ .valor = .{ .edad = 30, .altura = 175 } };
31	p.describir();
32	
33	const n = Serializable(i32){ .valor = 42 };
34	n.describir();
35	}

```

1      }
2      std.debug.print("{s}({\n", .{@typeName(T)});
3      inline for (info.@"struct".fields) |field| {
4          std.debug.print("    {s} = {},\n",
5              .{field.name, @field(self.valor, field.name)})
6      }
7      std.debug.print(")\n", .{});

```

@field combina introspección con acceso dinámico a campos.

APRENDE ZIG

Si vienes de otros lenguajes de programación, es útil comparar como Zig maneja la generación de código en tiempo de compilación frente a alternativas populares. Cada lenguaje tiene su propio enfoque, y entender las diferencias te ayudara a apreciar las ventajas de comptime.

Lenguaje	Mecanismo	Características
C/C++	Macros (#define)	Textual, no type-safe
C++	Templates	Potente, pero errores crípticos
Rust	Macros + Generics	Dos sistemas separados
Zig	comptime	Un solo lenguaje unificado

Tabla 50 - Comptime vs Otras tecnologías
(Fuente: Propia)

En C y C++, las macros del preprocesador realizan sustitución textual antes de la compilación. Esto es rápido pero propenso a errores: las macros no entienden tipos, no respetan scopes, y pueden producir bugs sutiles.

Los templates de C++ son más potentes y type-safe, pero tienen una sintaxis completamente diferente al código normal y producen mensajes de error notoriamente difíciles de entender.

1	<pre>// En C++, los templates tienen sintaxis especial:\ntemplate<typename T>\nT multiplicar(T a, T b) { return a * b; }\n\n// En Zig, es solo una función normal:\nfn multiplicar(comptime T: type, a: T, b: T) T {\n return a * b;\n}</pre>
---	---

Rust tiene un sistema dual: macros para metaprogramación textual y genéricos para código polimórfico. Los genéricos de Rust son potentes, pero requieren anotaciones de traits.

Para quienes conocen otros lenguajes, las macros procedurales de Rust son flexibles, pero operan en árboles de sintaxis, no en valores semánticos.

La ventaja principal de comptime en Zig es la unificación: el mismo código que escribes para tiempo de ejecución funciona en tiempo de compilación. No hay sintaxis especial para macros o templates. Si sabes escribir una función en Zig, ya sabes escribir código comptime.

Además, los errores de compilación son claros porque el compilador entiende exactamente que estás intentando hacer.



Consejo

Cuando aprendas un nuevo lenguaje, pregúntate: ¿cómo maneja la metaprogramación? La respuesta revela mucho sobre la filosofía del lenguaje. Zig eligió simplicidad y consistencia sobre características especializadas.

Aunque comptime es muy poderoso, tiene limitaciones importantes que debes conocer. Entender qué NO puedes hacer en comptime es tan importante como saber qué puedes hacer.

Estas limitaciones existen por buenas razones: garantizan que la compilación sea determinística, reproducible y segura.

Operación	Permitida	Razón
Leer archivos	No	Compilación no determinística
Acceder a red	No	Efectos secundarios externos
Memoria dinámica	Limitada	Solo con allocator comptime
Punteros a runtime	No	No existen en compilación
Llamar funciones inline asm	No	Requiere ejecución real

Tabla 51 - Limitaciones de comptime
(Fuente: Propia)

La primera y más importante limitación es que comptime no puede realizar operaciones de entrada/salida. No puedes leer archivos, acceder a la red, o interactuar con hardware durante la compilación.

Esto garantiza que compilar el mismo código dos veces produzca el mismo resultado, una propiedad esencial para builds reproducibles.

1

```
// Esto NO funciona - no puedes leer archivos en comptime:\nconst data =\ncomptime blk: {\n  // ERROR: std.fs no disponible en comptime\n  const\n  file = std.fs.cwd().openFile(\"config.txt\", .{});\n  break :blk\n  file.readAll();\n};\n\n// En cambio, usa @embedFile para incluir archivos:\nconst data = @embedFile(\"config.txt\");
```

La segunda limitación es sobre punteros. En tiempo de compilación, la memoria de runtime no existe todavía. No puedes tener un puntero a una variable que solo existirá cuando el programa se ejecute.

Sin embargo, puedes trabajar con valores directamente y el compilador los manejará apropiadamente



Una limitación práctica es la recursión. Aunque puedes llamar funciones recursivamente en comptime, hay un límite en la profundidad para evitar compilaciones infinitas. Si alcanzas este límite, necesitas reestructurar tu código o usar un enfoque iterativo.

1

```
// Recursión comptime tiene límites:\nfn factorial(comptime n: u64) u64
{\n    if (n ≤ 1) return 1;\n    return n * factorial(n - 1);\n}\n\nconst
f10 = factorial(10); // OK: 3628800\nconst f20 = factorial(20); // OK:
2432902008176640000\n// const f1000 = factorial(1000); // Podría exceder
límites
```

Ahora que entiendes tanto las capacidades como las limitaciones de comptime, exploremos algunos casos de uso avanzados que demuestran su poder. Estos ejemplos representan patrones que encontrarás en código Zig real y en la librería estándar.

Un caso de uso muy común es crear interfaces para diferentes backends. Por ejemplo, puedes escribir código que funcione con diferentes implementaciones de logging, bases de datos, o protocolos de red, seleccionando la implementación en tiempo de compilación:

1

```
fn Logger(comptime backend: enum { console, file, network }) type {\n
return struct {\n    pub fn log(msg: []const u8) void {\n
switch (backend) {\n    .console => std.debug.print("\{s}\n",
.\{msg}\},\n    .file => { /* escribir a archivo */ },\n
.network => { /* enviar por red */ },\n    }\n    }\n    };\n\n
\n\nconst MyLogger = Logger(.console);\nMyLogger.log("Mensaje de
prueba");
```

Otro patrón avanzado es la generación de estructuras de datos optimizadas. Puedes crear estructuras adaptadas a los datos específicos que manejarás:

1

```
fn BitSet(comptime N: usize) type {\n    const num_words = (N + 63) /
64;\n    return struct {\n        bits: [num_words]u64,\n\n        pub fn
init() @This() {\n            return .{ .bits = [_]u64{0} ** num_words };
\n        }\n\n        pub fn set(self: *@This(), index: usize) void {\n
const word = index / 64;\n            const bit = @as(u6, @intCast(index %
64));\n            self.bits[word] |= (@as(u64, 1) << bit);\n        }\n
\n        pub fn get(self: *const @This(), index: usize) bool {\n            const
word = index / 64;\n            const bit = @as(u6, @intCast(index % 64));
\n            return (self.bits[word] & (@as(u64, 1) << bit)) ≠ 0;\n
        }\n    };\n\n\nvar bits = BitSet(1000).init(); // BitSet para 1000
bits\nbits.set(42);
```

La librería estándar de Zig usa extensivamente comptime para implementar estructuras de datos genéricas. Por ejemplo, `std.ArrayList` es una función que retorna un tipo, similar a los ejemplos que hemos visto. Cuando escribes `std.ArrayList(u8)`, comptime genera un tipo específico para arrays de bytes.

Estudiar la librería estándar de Zig es una excelente manera de aprender patrones avanzados de comptime. El código está bien documentado y muestra usos idiomáticos del lenguaje.

Finalmente, comptime es fundamental para crear DSLs (Domain-Specific Languages, un mini-lenguaje diseñado para un dominio específico, por ejemplo, el lenguaje de consultas SQL es un DSL para bases de datos) embebidos. Puedes procesar strings en tiempo de compilación para crear parsers, validadores de formato, o incluso mini-lenguajes personalizados:

1

```
fn parseFormat(comptime fmt: []const u8) usize {\n    var count: usize = 0;\n    for (fmt) |c| {\n        if (c == '{' and fmt[count + 1] == '{')\n            count += 1;\n    }\n    return count;\n}\n\n// std.debug.print usa comptime para validar formato:\nstd.debug.print("{} + {} = {}\\n", .{1, 2, 3}); // OK\nstd.debug.print("{} + {}\\n", .{1, 2, 3}); // Error: demasiados args
```

En este capítulo exploraste comptime y la metaprogramación en Zig. Aprendiste que comptime permite ejecutar código durante la compilación, produciendo valores y tipos que están disponibles sin costo en tiempo de ejecución.

Viste como los parámetros comptime permiten crear funciones genéricas que funcionan con diferentes tipos, y como las funciones pueden retornar tipos para construir estructuras de datos personalizadas. Los bloques comptime te permiten realizar cálculos complejos antes de que el programa se ejecute.

En el próximo capítulo aprenderemos sobre entrada/salida y archivos, aplicando todo lo que sabes para interactuar con el sistema de archivos y el mundo exterior.

EJERCICIOS

Pon a prueba tus conocimientos con los siguientes ejercicios

TABLA DE POTENCIAS

Crea una tabla con las primeras 10 potencias de 2 (2^0 hasta 2^9).
Luego escribe código que use esta tabla para imprimir los valores.

Ejercicio

CAPÍTULO 12

Entrada/Salida y Archivos

Los programas raramente existen en aislamiento. Necesitan leer datos de archivos, escribir resultados, recibir entrada del usuario, y comunicarse con el mundo exterior. Estas operaciones de entrada/salida (E/S o I/O del inglés) son fundamentales para cualquier programa útil. En este capítulo aprenderás como Zig maneja la E/S, desde la lectura de archivos de texto hasta la escritura de datos binarios.

Stream	Función	Uso
stdin	std.io.getStdIn()	Entrada del usuario
stdout	std.io.getStdOut()	Salida normal
stderr	std.io.getStdErr()	Mensajes de error

Tabla 52 - Streams Estándar
(Fuente: Propia)

La filosofía de E/S de Zig refleja los principios del lenguaje: control explícito, manejo de errores robusto, y sin costos ocultos. A diferencia de lenguajes de alto nivel que esconden muchos detalles, Zig te muestra exactamente que está sucediendo. Esto puede parecer más verboso al principio, pero te da control completo y evita sorpresas desagradables.

Empezaremos con la salida estándar que ya has estado usando, luego exploraremos la entrada del usuario, y finalmente nos adentraremos en el sistema de archivos. Al final del capítulo, serás capaz de escribir programas que interactúan con el sistema de archivos de manera segura y eficiente.

Ya has estado usando `std.debug.print` para mostrar información en la consola. Esta función es conveniente para depuración, pero para programas de producción es mejor usar los streams estándar directamente. La salida estándar (`stdout`) es el canal principal para la salida del programa:

1	<code>const std = @import("std");</code>
2	
3	<code>pub fn main() !void {</code>
4	<code> const stdout = std.io.getStdOut().writer();</code>
5	<code> try stdout.print(";Hola, mundo!\n", .{});</code>
6	<code> try stdout.print("El número es: {}\n", .{42});</code>
7	<code>}</code>

`std.io.getStdOut()` obtiene el descriptor de archivo de la salida estándar, y `.writer()` crea un escritor (writer) que puedes usar para enviar datos. El método `print` funciona de manera similar a `std.debug.print`, con formato y argumentos.

Observa que usamos `try` porque las operaciones de E/S pueden fallar. Por ejemplo, si el programa se ejecuta con la salida redirigida a un archivo y el disco está lleno, la escritura fallaría. Zig te obliga a manejar esta posibilidad.

NOTA

`std.debug.print` es para depuración y escribe a `stderr` (error estándar). Para la salida normal del programa, usa `stdout`. Para mensajes de error, usa `stderr`. Esta distinción es importante porque permite que los usuarios redirijan la salida normal sin mezclarla con mensajes de error.

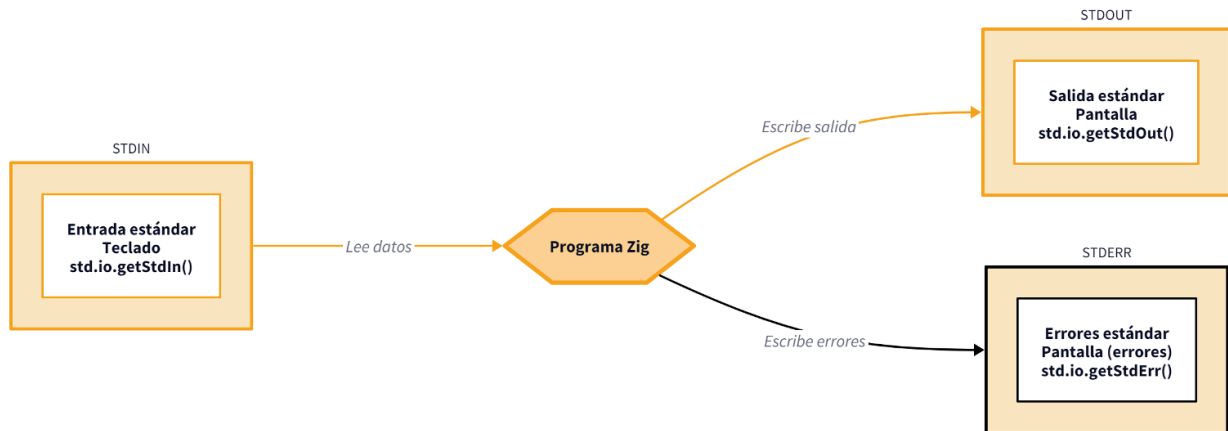


Figura 21 - Streams estándar

La entrada estándar (stdin) es el canal por donde el programa recibe datos del usuario. Leer del stdin es un poco más complejo porque necesitas un buffer donde almacenar los datos leídos:

1	<code>const std = @import("std");</code>
2	
3	<code>pub fn main() !void {</code>
4	<code> const stdin = std.io.getStdIn().reader();</code>
5	<code> const stdout = std.io.getStdOut().writer();</code>
6	
7	<code> try stdout.print("¿Como te llamas? ", .{});</code>
8	
9	<code> var buffer: [100]u8 = undefined;</code>
10	<code> const input = stdin.readUntilDelimiter(&buffer, '\n') catch err {</code>
11	<code> if (err == error.EndOfStream) return;</code>
12	<code> return err;</code>
13	<code> };</code>

14	
15	<code>try stdout.print(";Hola, {s}!\n", .{input});</code>
16	<code>}</code>

El reader proporciona varios métodos para leer datos. `readUntilDelímiter` lee hasta encontrar un carácter específico (en este caso, nueva línea). El resultado es un slice que apunta al buffer, conteniendo solo los caracteres leídos sin incluir el delimitador.

Es importante manejar el caso donde el usuario cierra la entrada (Ctrl+D en Linux, Ctrl+Z en Windows), lo cual produce `error.EndOfStream`. El buffer tiene un tamaño fijo, así que, si la línea es más larga, se truncará o producirá un error según el método que uses.

Para trabajar con archivos, primero debes abrirlos. Zig proporciona `std.fs.cwd()` para acceder al directorio actual de trabajo, y desde ahí puedes abrir archivos:

1	<code>const std = @import("std");</code>
2	
3	<code>pub fn main() !void {</code>
4	<code> // Abrir un archivo para lectura</code>
5	<code> const archivo = try std.fs.cwd().openFile("datos.txt", .{});</code>
6	<code> defer archivo.close(); // Siempre cerrar al terminar</code>
7	
8	<code> // Ahora puedes leer del archivo...</code>
9	<code>}</code>

`openFile` toma el nombre del archivo y una estructura de opciones. Con opciones vacías (`.{}`), abre el archivo solo para lectura. El `defer archivo.close()` es crucial: asegura que el archivo se cierre cuando la función termine, incluso si hay un error más adelante.

Para escribir en un archivo, necesitas especificar qué quieres acceso de escritura y posiblemente crear el archivo si no existe:

1	<code>// Abrir (o crear) un archivo para escritura</code>
2	<code>const archivo = try std.fs.cwd().createFile("salida.txt", .{});</code>
3	<code>defer archivo.close();</code>

4	
5	// Para leer Y escribir:
6	const archivo_rw = try std.fs.cwd().openFile("datos.txt", .{ .mode = .read_write });



Consejo

Siempre usa `defer` para cerrar archivos inmediatamente después de abrirlos. Esto garantiza que el archivo se cierre correctamente sin importar como termine la función (retorno normal, error, etc.).

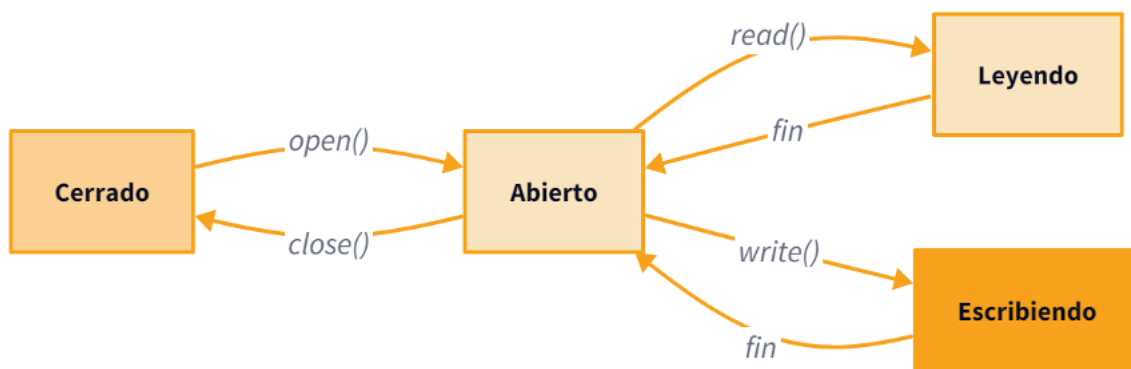


Figura 22 - Ciclo de vida de un archivo

Una vez que tienes un archivo abierto, puedes leerlo de varias formas. La más simple es leer todo el contenido de una vez, aunque esto solo es apropiado para archivos pequeños:

Una vez que tienes un archivo abierto, puedes leerlo de varias formas. La más simple es leer todo el contenido de una vez, aunque esto solo es apropiado para archivos pequeños:

1	<code>const std = @import("std");</code>
2	
3	<code>pub fn main() !void {</code>
4	<code>var gpa = std.heap.GeneralPurposeAllocator(.{}){};</code>
5	<code>defer _ = gpa.deinit();</code>
6	<code>const allocator = gpa.allocator();</code>
7	
8	<code>const archivo = try std.fs.cwd().openFile("datos.txt", .{});</code>
9	<code>defer archivo.close();</code>
10	
11	<code>// Leer todo el archivo (maximo 1MB)</code>
12	<code>const contenido = try archivo.reader().readAllAlloc(allocator, 1024 * 1024);</code>
13	<code>defer allocator.free(contenido);</code>
14	
15	<code>std.debug.print("Contenido:\n{s}\n", .{contenido});</code>
16	<code>}</code>

El método `readAllAlloc` usa un `allocator` para reservar memoria y lee el archivo completo. El segundo parámetro es el tamaño máximo permitido, para evitar llenar la memoria con archivos enormes. Recuerda liberar la memoria con `defer`.

Para archivos grandes, es mejor leer línea por línea o en chunks. Esto es más eficiente en memoria:

1	<code>pub fn main() !void {</code>
2	<code>const archivo = try std.fs.cwd().openFile("datos.txt", .{});</code>
3	<code>defer archivo.close();</code>
4	
5	<code>var buffer: [1024]u8 = undefined;</code>

6	<code>const reader = archivo.reader();</code>
7	
8	<code>var línea_num: usize = 0;</code>
9	<code>while (reader.readUntilDelimiter(&buffer, '\n')) línea {</code>
10	<code> línea_num += 1;</code>
11	<code> std.debug.print("{}: {}{}\n", .{línea_num, línea});</code>
12	<code> } else err {</code>
13	<code> if (err != error.EndOfStream) return err;</code>
14	<code> }</code>
15	<code>}</code>

Este patrón lee cada línea en un buffer de tamaño fijo. El bucle while con else maneja el final del archivo: cuando no hay más líneas, readUntilDelimiter retorna error.EndOfStream, que capturamos y tratamos como terminación normal.

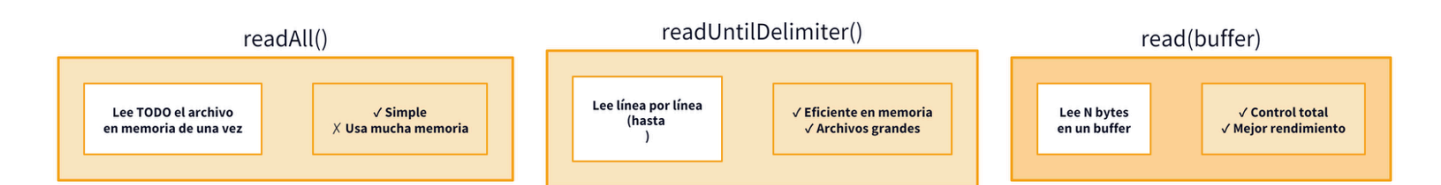


Figura 23 - Estrategias de lectura

Escribir en archivos es similar a escribir en stdout. Primero creas o abres un archivo con permisos de escritura, luego usas el writer:

1	<code>const std = @import("std");</code>
2	
3	<code>pub fn main() !void {</code>
4	<code>const archivo = try std.fs.cwd().createFile("salida.txt", .{});</code>
5	<code>defer archivo.close();</code>
6	
7	<code>const writer = archivo.writer();</code>
8	
9	<code>try writer.print("Línea 1\n", .{});</code>
10	<code>try writer.print("El valor es: {}\n", .{42});</code>
11	<code>try writer.print("Nombre: {s}\n", .{"Zig"});</code>
12	
13	<code>// También puedes escribir bytes directamente</code>
14	<code>try writer.writeAll("Datos finales\n");</code>
15	<code>}</code>

El método `print` funciona igual que para `stdout`, con formato y argumentos. El método `writeAll` escribe bytes directamente sin formato. Hay también `writeByte` para escribir un solo byte, y otros métodos para diferentes necesidades.

Para mejorar el rendimiento, especialmente cuando haces muchas operaciones pequeñas de E/S, puedes usar escritores y lectores con buffer. Estos acumulan los datos en memoria y hacen llamadas al sistema operativo menos frecuentes:

1	<code>const archivo = try std.fs.cwd().createFile("salida.txt", .{});</code>
2	<code>defer archivo.close();</code>
3	
4	<code>var buffered = std.io.bufferedWriter(archivo.writer());</code>
5	<code>const writer = buffered.writer();</code>
6	
7	<code>// Muchas escrituras pequeñas</code>
8	<code>for (0..1000) i {</code>

9	try writer.print("Línea {}\n", .{i});
10	}
11	
12	// Importante: vaciar el buffer al final
13	try buffered.flush();

NOTA

Cuando usas E/S con buffer, es crítico llamar a flush() al final para asegurar que todos los datos se escriban. Si el archivo se cierra antes de hacer flush, podrías perder los últimos datos que quedaron en el buffer.

Además de leer y escribir archivos, frecuentemente necesitas realizar otras operaciones como crear directorios, eliminar archivos, o verificar si un archivo existe. Zig proporciona estas funciones a través de std.fs:

1	const std = @import("std");
2	
3	pub fn main() !void {
4	const cwd = std.fs.cwd();
5	
6	// Crear un directorio
7	cwd.mkdir("nuevo_dir") catch err {
8	if (err != error.PathAlreadyExists) return err;
9	};
9	
10	

11	// Renombrar un archivo
12	try cwd.rename("viejo.txt", "nuevo.txt");
13	
14	// Eliminar un archivo
15	try cwd.deleteFile("temporal.txt");
16	
17	// Eliminar un directorio vacío
18	try cwd.deleteDir("dir_vacío");
19	}

Para verificar si un archivo existe, puedes intentar abrir y manejar el error si no existe, o usar `access` para verificar sin abrir:

1	fn archivo_existe(nombre: []const u8) bool {
2	std.fs.cwd().access(nombre, .{}) catch return false;
3	return true;
4	}
5	
6	if (archivo_existe("config.txt")) {
7	std.debug.print("El archivo existe\n", .{});
8	}

Operación	Método	Propósito
Crear directorio	<code>makeDir()</code>	Nueva carpeta
Renombrar	<code>rename()</code>	Cambiar nombre
Eliminar archivo	<code>deleteFile()</code>	Borrar archivo
Verificar acceso	<code>access()</code>	Comprobar existencia

Tabla 53 - Operaciones del Sistema de Archivos
(Fuente: Propia)

Trabajar con rutas de archivos puede ser complicado debido a las diferencias entre sistemas operativos. Windows usa barras invertidas y letras de unidad, mientras que Linux y macOS usan barras normales. Zig proporciona `std.fs.path` para manejar rutas de forma portable:

1	<code>const std = @import("std");</code>
2	
3	<code>pub fn main() !void {</code>
4	<code>var gpa = std.heap.GeneralPurposeAllocator(.{}){};</code>
5	<code>defer _ = gpa.deinit();</code>
6	<code>const allocator = gpa.allocator();</code>
7	
8	<code>// Unir componentes de ruta</code>
10	<code>const ruta = try std.fs.path.join(allocator, &.{ "dir", "subdir", "archivo.txt" });</code>
11	
12	<code>std.debug.print("Ruta: {s}\n", .{ruta});</code>
13	
14	<code>// Obtener el nombre del archivo sin la ruta</code>
15	<code>const nombre = std.fs.path.basename("/home/user/documento.txt");</code>
16	<code>std.debug.print("Nombre: {s}\n", .{nombre});</code>
17	
18	<code>// Obtener el directorio padre</code>
19	<code>const dir = std.fs.path.dirname("/home/user/documento.txt");</code>
20	<code>std.debug.print("Directorio: {s}\n", .{dir orelse "(ninguno)"});</code>
21	<code>}</code>

Los programas frecuentemente reciben parámetros a través de argumentos de línea de comandos. Zig proporciona `std.process.args()` para acceder a ellos:

1	<code>const std = @import("std");</code>
2	

3	pub fn main() !void {
4	var gpa = std.heap.GeneralPurposeAllocator(.{}){};
5	defer _ = gpa.deinit();
6	const allocator = gpa.allocator();
7	
8	var args = try std.process.argsWithAllocator(allocator);
9	defer args.deinit();
10	
11	// El primer argumento es el nombre del programa
12	const programa = args.next() orelse "desconocido";
13	std.debug.print("Programa: {s}\n", .{programa});
14	
15	// El resto son argumentos del usuario
16	while (args.next()) arg {
17	std.debug.print("Argumento: {s}\n", .{arg});
18	}
19	}

El primer argumento siempre es el nombre o ruta del programa. Los siguientes son los argumentos que el usuario proporcionó. Este patrón es la base para crear herramientas de línea de comandos.

Veamos un programa completo que lee un archivo de texto, cuenta las líneas, palabras y caracteres, y escribe un resumen en otro archivo:

1	const std = @import("std");
2	
3	pub fn main() !void {
4	defer _ = gpa.deinit();
5	const allocator = gpa.allocator();
6	const allocator = gpa.allocator();

7	
8	<code>// Abrir archivo de entrada</code>
9	<code>const entrada = std.fs.cwd().openFile("texto.txt", .{}) catch err {</code>
10	<code> std.debug.print("Error abriendo archivo: {s}\n", .</code> <code>{@errorName(err)});</code>
11	<code> return;</code>
12	<code>};</code>
13	<code>defer entrada.close();</code>
14	
15	<code>// Leer contenido</code>
16	<code>const contenido = try entrada.reader().readAllAlloc(allocator, 1024 * 1024);</code>
17	<code>defer allocator.free(contenido);</code>
18	
19	<code>// Contar estadísticas</code>
20	<code>var líneas: usize = 0;</code>
21	<code>var palabras: usize = 0;</code>
22	<code>var en_palabra = false;</code>
23	
24	<code>for (contenido) c {</code>
25	<code> if (c == '\n') líneas += 1;</code>
26	<code> if (c == ' ' or c == '\n' or c == '\t') {</code>
27	<code> en_palabra = false;</code>
28	<code> } else {</code>
29	<code> if (!en_palabra) palabras += 1;</code>
30	<code> en_palabra = true;</code>
31	<code> }</code>
32	<code>}</code>
33	

34	// Escribir resultado
35	const salida = try std.fs.cwd().createFile("estadisticas.txt", .{});
36	defer salida.close();
37	
38	const writer = salida.writer();
39	try writer.print("Estadísticas del archivo:\n", .{});
39	defer entrada.close();
40	try writer.print("Líneas: {}\n", .{líneas});
41	try writer.print("Palabras: {}\n", .{palabras});
42	try writer.print("Caracteres: {}\n", .{contenido.len});
43	
44	std.debug.print("Estadísticas guardadas en estadisticas.txt\n", .{});
45	}

The screenshot shows a code editor window titled "estadisticas_archivo.zig". The code is as follows:

```

1  const salida = try std.fs.cwd().createFile("estadisticas.txt",
2      defer salida.close();
3      const writer = salida.writer();
4      try writer.print("Estadísticas del archivo:\n", .{});
5      try writer.print("Líneas: {}\n", .{líneas});
6      try writer.print("Palabras: {}\n", .{palabras});
7      try writer.print("Caracteres: {}\n", .{contenido.len});
8  }
9  // Fin del programa

```

At the bottom of the editor, there is a message: "Este patrón evita contar espacios consecutivos como palabras." Below that, the text "APRENDE ZIG" is visible.

Más allá de las operaciones básicas de archivos, Zig ofrece capacidades avanzadas para trabajo con I/O. Estos conceptos son útiles para aplicaciones más complejas que manejan grandes volúmenes de datos o requieren alto rendimiento.

Concepto	Descripción	Uso Típico
Buffered I/O	Reduce syscalls	Archivos grandes
Memory-mapped files	Acceso directo a memoria	Archivos enormes
Asynchronous I/O	Operaciones no bloqueantes	Servidores
Polling	Múltiples fuentes	Multiplexación

Tabla 54 - Conceptos Avanzados de I/O
(Fuente: Propia)

El `BufferedReader` y `BufferedWriter` reducen el número de llamadas al sistema agrupando operaciones pequeñas. Esto es especialmente importante cuando lees o escribes muchos datos pequeños:

1	<code>const std = @import("std");</code>
2	
3	<code>pub fn main() !void {</code>
4	<code>const file = try std.fs.cwd().createFile("salida.txt", .{});</code>
5	<code>defer file.close();</code>
6	
7	<code>// Writer con buffer de 4KB - mucho más eficiente</code>
8	<code>var buf_writer = std.io.bufferedWriter(file.writer());</code>
9	<code>const writer = buf_writer.writer();</code>
10	
11	<code>// Escribir muchas líneas pequeñas</code>
12	<code>for (0..1000) i {</code>

13	<code>try writer.print("Línea {}\n", .{i});</code>
14	<code>}</code>
15	<code>// Asegurar que todo se escriba</code>
16	<code>try buf_writer.flush();</code>
17	<code>}</code>

Para archivos muy grandes, considera usar el módulo `std.os.mmap` para mapear archivos directamente a memoria. Esto permite tratar un archivo como si fuera un array en memoria, dejando que el sistema operativo maneje la paginación.



Consejo

Cuando trabajes con archivos grandes (>100MB), siempre usa operaciones buffereadas o memory-mapped files. La diferencia de rendimiento puede ser de 10x o más comparado con lecturas/escrituras individuales.

Los programas frecuentemente necesitan guardar y leer datos estructurados. Hay varios enfoques para serializar datos en archivos:

Operación	Método	Propósito
Texto plano	Simple, legible	Parsing complejo, ineficiente
CSV	Tabular, portable	Sin tipos, escapes problemáticos
JSON	Estructurado, universal	Verbose, parsing lento
Binario	Compacto, rápido	No legible, versionado difícil

Tabla 55 - Formatos de Datos Comunes
(Fuente: Propia)

Para formatos binarios simples, puedes usar `std.mem` para convertir estructuras a bytes y viceversa:

1	<code>const std = @import("std");</code>
2	<code>const Registro = packed struct {</code>
3	<code> id: u32,</code>
4	<code> valor: f32,</code>
5	<code> activo: bool,</code>
6	<code>};</code>
7	<code>fn guardar_registro(file: std.fs.File, reg: Registro) !void {</code>
8	<code> const bytes = std.mem.asBytes(&reg);</code>
9	<code> try file.writer().writeAll(bytes);</code>
10	<code>fn leer_registro(file: std.fs.File) !Registro {</code>
11	<code> // Escribir muchas líneas pequeñas</code>
12	<code> _ = try file.reader().readAll(&bytes);</code>
13	<code> return std.mem.bytesToValue(Registro, &bytes);</code>
14	<code>}</code>

Advertencia

Al usar formatos binarios, ten cuidado con la portabilidad entre plataformas. El endianness (orden de bytes) puede variar. Para datos que se compartirán entre sistemas, considera usar formatos con endianness explícito o texto.

En este capítulo aprendiste a realizar operaciones de entrada/salida en Zig. Viste como usar la salida estándar (stdout) para mostrar información y la entrada estándar (stdin) para leer datos del usuario.

Exploramos como abrir, leer y escribir archivos usando la librería estándar. Aprendiste la importancia de usar defer para cerrar archivos, como leer archivos completos o línea por línea, y como usar escritores con buffer para mejor rendimiento.

También viste operaciones del sistema de archivos como crear directorios, renombrar y eliminar archivos. Aprendiste a manipular rutas de forma portable y a procesar argumentos de línea de comandos.

En el próximo y último capítulo, pondremos en práctica todo lo aprendido en un proyecto final integrador que combinará estructuras, manejo de errores, E/S, y las demás técnicas que has dominado a lo largo del libro.

EJERCICIOS

Pon a prueba tus conocimientos con los siguientes ejercicios

ECO DE LÍNEA

Escribe un programa que lea una línea de texto del usuario y la imprima de vuelta precedida por 'Escribiste: '. Usa stdin y stdout correctamente.

Ejercicio

1

ESCRIBIR LISTA A ARCHIVO

Crea un programa que escriba los números del 1 al 100 en un archivo llamado 'números.txt', uno por línea.

Ejercicio

2

CONTAR LÍNEAS

Escribe un programa que lea un archivo y cuente cuantas líneas tiene. El nombre del archivo debe pasarse como argumento de línea de comandos.

Ejercicio

3

CAPÍTULO 13

Proyecto final integrador

Has llegado al capítulo final de este libro, y es momento de poner en práctica todo lo que has aprendido. A lo largo de los capítulos anteriores, has explorado variables y tipos de datos, estructuras de control, funciones, arrays y slices, punteros y memoria, estructuras personalizadas, manejo de errores, comptime, y operaciones de entrada/salida. Ahora, vamos a combinar todas estas habilidades en un proyecto real y completo.

El proyecto que construiremos es un gestor de tareas de línea de comandos. Esta aplicación permitirá a los usuarios agregar tareas, marcarlas como completadas, listarlas con diferentes filtros, y guardarlas en un archivo para persistencia.

Construiremos el proyecto paso a paso, explicando las decisiones de diseño y mostrando como los conceptos que has aprendido se aplican en un contexto real.

Antes de escribir código, es importante planificar la estructura del proyecto. Nuestro gestor de tareas necesita varias capacidades: representar tareas con título y estado, almacenar múltiples tareas, leer y escribir tareas a un archivo, y procesar comandos del usuario.

Usaremos estructuras para representar las tareas y el estado de la aplicación. Los enums nos ayudarán a definir las prioridades y estados posibles. El manejo de errores garantizará que el programa sea robusto. Y las operaciones de E/S permitirán guardar y cargar las tareas desde un archivo.

Comando	Argumento	Acción
agregar	<título>	Añade nueva tarea
listar	(opcional filtro)	Muestra tareas
completar	<id>	Marca como terminada
eliminar	<id>	Borra la tarea

Tabla 56 - Comandos del Gestor de Tareas
(Fuente: Propia)

NOTA

En proyectos reales, esta fase de diseño puede tomar más tiempo que la codificación misma. Pensar en la estructura antes de escribir código evita muchos problemas y reescrituras posteriores.

Comencemos definiendo las estructuras de datos principales. Una tarea tendrá un identificador único, un título, una prioridad, y un estado que indica si esta completada:

1	<code>const std = @import("std");</code>
2	
3	<code>const Prioridad = enum {</code>
4	<code> baja,</code>
5	<code> media,</code>
6	<code> alta,</code>
7	
8	<code>pub fn símbolo(self: Prioridad) u8 {</code>
9	<code> return switch (self) {</code>
10	<code> .baja => '.',</code>
11	<code> return;</code>
12	<code> .alta => '!',</code>
13	<code> };</code>
14	<code>}</code>
15	<code>// Leer contenido</code>
16	
17	<code>const Tarea = struct {</code>

18	id: u32,
19	titulo: [100]u8,
20	titulo_len: usize,
21	prioridad: Prioridad,
22	completada: bool,
23	
24	pub fn obtener_titulo(self: *const Tarea) []const u8 {
25	return self.titulo[0..self.titulo_len];
26	}
27	};



El enum Prioridad define tres niveles con un método para obtener un símbolo visual. La estructura Tarea usa un array de tamaño fijo para el título junto con una longitud, lo que evita la necesidad de un allocator para cada tarea individual. El método obtener_titulo retorna un slice con solo los caracteres validos del título.

Ahora creemos una estructura que gestione la colección de tareas. Esta estructura será responsable de agregar, buscar, y modificar tareas:

1	const MAX_TAREAS = 100;
2	const GestorTareas = struct {
3	tareas: [MAX_TAREAS]Tarea,
4	cantidad: usize,
5	siguiente_id: u32,
6	pub fn init() GestorTareas {
7	return .{
8	.tareas = undefined,
9	.cantidad = 0,
10	.siguiente_id = 1,
11	};
12	}
13	pub fn agregar(self: *GestorTareas, titulo: []const u8, prio: Prioridad) !u32 {
14	if (self.cantidad ≥ MAX_TAREAS) {
15	return error.ListaLlena;
16	}
17	if (titulo.len == 0 or titulo.len > 100) {
18	return error.TituloInvalido;
19	}
20	const id = self.siguiente_id;
21	var tarea = &self.tareas[self.cantidad];
22	tarea.id = id;
23	tarea.titulo_len = titulo.len;

24	@memcpy(tarea.titulo[0..titulo.len], titulo);
25	tarea.prioridad = prio;
26	tarea.completada = false;
27	self.cantidad += 1;
28	self.siguiente_id += 1;
29	return id;
30	}
31	};

El gestor usa un array de tamaño fijo para almacenar las tareas. Esto simplifica la gestión de memoria al no requerir un allocator, aunque limita el número de tareas. El método agregar valida los datos de entrada y retorna errores si algo está mal, siguiendo el patrón de manejo de errores que aprendiste.



Figura 24 - Diagrama de struct Tarea

Agreguemos métodos para buscar tareas por ID y modificar su estado:

1	pub fn buscar_por_id(self: *GestorTareas, id: u32) ?*Tarea {
2	for (self.tareas[0..self.cantidad]) *tarea {
3	if (tarea.id == id) return tarea;

4	}
5	return null;
6	}
7	
8	pub fn completar(self: *GestorTareas, id: u32) !void {
9	const tarea = self.buscar_por_id(id) orelse return error.TareaNoEncontrada;
10	tarea.completada = true;
11	}
12	
13	pub fn eliminar(self: *GestorTareas, id: u32) !void {
14	for (self.tareas[0..self.cantidad], 0..) *tarea, i {
15	if (tarea.id == id) {
16	// Mover la ultima tarea a esta posicion
17	if (i < self.cantidad - 1) {
18	self.tareas[i] = self.tareas[self.cantidad - 1];
19	}
20	self.cantidad -= 1;
21	return;
22	}
23	}
24	return error.TareaNoEncontrada;
25	}

El método `buscar_por_id` retorna un puntero opcional a la tarea, permitiendo modificarla directamente. El método `completar` usa este patrón para marcar una tarea como terminada. El método `eliminar` mueve la última tarea al hueco para mantener el array compacto, un truco común para eliminar elementos rápidamente cuando el orden no importa.

Un método esencial es listar las tareas. Implementaremos filtros para ver todas las tareas, solo las pendientes, o solo las completadas:

1	const Filtro = enum { todas, pendientes, completadas };
2	pub fn listar(self: *GestorTareas, filtro: Filtro, writer: anytype) ! void {
3	var mostradas: usize = 0;
4	for (self.tareas[0..self.cantidad]) *tarea {
5	const mostrar = switch (filtro) {
6	.todas ⇒ true,
7	.pendientes ⇒ !tarea.completada,
8	.completadas ⇒ tarea.completada,
9	};
10	if (mostrar) {
11	const estado: u8 = if (tarea.completada) 'x' else ' ';
12	try writer.print("[{c}] {c} #{i}: {s}\n", .{
13	estado,
14	tarea.prioridad.símbolo(),
15	tarea.id,
16	tarea.obtener_titulo(),
17	});
18	mostradas += 1;
19	}
20	}
21	if (mostradas == 0) {
22	try writer.print("(no hay tareas)\n", .{});
23	}
24	}



Consejo

Usar 'anytype' como tipo del writer permite que este método funcione tanto con stdout como con un archivo o cualquier otro writer. Es un ejemplo de código genérico sin necesidad de parámetros comptime explícitos.

Para que las tareas persistan entre ejecuciones, necesitamos guardarlas y cargarlas desde un archivo. Usaremos un formato de texto simple donde cada línea representa una tarea:

1	<code>const Filtro = enum { todas, pendientes, completadas };</code>
2	<code>pub fn listar(self: *GestorTareas, filtro: Filtro, writer: anytype) !void {</code>
3	<code>var mostradas: usize = 0;</code>
4	<code>for (self.tareas[0..self.cantidad]) *tarea {</code>
5	<code>const mostrar = switch (filtro) {</code>
6	<code>.todas => true,</code>
7	<code>.pendientes => !tarea.completada,</code>
8	<code>.completadas => tarea.completada,</code>
9	<code>};</code>
10	<code>if (mostrar) {</code>
11	<code>const estado: u8 = if (tarea.completada) 'x' else ' ';</code>
12	<code>try writer.print("[{c}] {c} #{i}: {s}\n", .{</code>
13	<code>estado,</code>
14	<code>tarea.prioridad.símbolo(),</code>

15	<code>tarea.id,</code>
16	<code>tarea.obtener_titulo(),</code>
17	<code>});</code>
18	<code>mostradas += 1;</code>
19	<code>}</code>
20	<code>}</code>
21	<code>if (mostradas == 0) {</code>
22	<code>try writer.print("(no hay tareas)\n", .{});</code>
23	<code>}</code>
24	<code>}</code>

El formato usa el carácter | como separador entre campos. La primera línea guarda el siguiente ID para mantener IDs únicos entre sesiones. Cada tarea ocupa una línea con su ID, estado de completado, prioridad, y título.

Ahora implementemos la carga desde archivo:

1	<code>pub fn cargar(self: *GestorTareas) !void {</code>
2	<code> const archivo = std.fs.cwd().openFile(ARCHIVO_TAREAS, .{}) catch </code> <code>err {</code>
3	<code> if (err == error.FileNotFound) return; // Sin archivo,</code> <code>empezar vacío</code>
4	<code> return err;</code>
5	<code> };</code>
6	<code> defer archivo.close();</code>
7	
8	<code> var buffer: [256]u8 = undefined;</code>
9	<code> const reader = archivo.reader();</code>
10	
11	<code> // Leer siguiente ID</code>
12	<code> if (reader.readUntilDelimiter(&buffer, '\n')) línea {</code>

13	<code>self.siguiente_id = std.fmt.parseInt(u32, línea, 10) catch 1;</code>
14	<code>} else _ return;</code>
15	
16	<code>// Leer tareas</code>
17	<code>while (reader.readUntilDelimiter(&buffer, '\n')) línea {</code>
18	<code>self.parsear_tarea(línea) catch continue;</code>
19	<code>} else _ {}</code>
20	<code>}</code>
21	
22	<code>fn parsear_tarea(self: *GestorTareas, línea: []const u8) !void {</code>
23	<code>var it = std.mem.splitScalar(u8, línea, ' ');</code>
24	
25	<code>const id_str = it.next() orelse return error.FormatoInvalido;</code>
26	<code>const compl_str = it.next() orelse return error.FormatoInvalido;</code>
27	<code>const prio_str = it.next() orelse return error.FormatoInvalido;</code>
28	<code>const titulo = it.next() orelse return error.FormatoInvalido;</code>
29	
30	<code>if (self.cantidad ≥ MAX_TAREAS) return error.ListaLlena;</code>
31	
32	<code>var tarea = &self.tareas[self.cantidad];</code>
33	<code>tarea.id = std.fmt.parseInt(u32, id_str, 10) catch return error.FormatoInvalido;</code>
34	<code>tarea.completada = compl_str.len > 0 and compl_str[0] == '1';</code>
35	<code>tarea.prioridad = switch (prio_str[0]) {</code>
36	<code>'A' ⇒ .alta,</code>
37	<code>'M' ⇒ .media,</code>
38	<code>else ⇒ .baja,</code>
39	<code>};</code>
40	<code>tarea.titulo_len = @min(titulo.len, 100);</code>

41	<code>@memcpy(tarea.titulo[0..tarea.titulo_len], titulo[0..tarea.titulo_len]);</code>
42	
43	<code>self.cantidad += 1;</code>
44	<code>}</code>

La función cargar primero verifica si el archivo existe. Si no existe, simplemente retorna sin error, ya que es normal empezar sin tareas. Luego lee línea por línea y usa `parsear_tarea` para interpretar cada una.

Finalmente, implementemos la función `main` que procesa comandos del usuario. El programa aceptará argumentos de línea de comandos para diferentes acciones:

1	<code>pub fn main() !void {</code>
2	<code>var gpa = std.heap.GeneralPurposeAllocator(.{}){};</code>
3	<code>defer _ = gpa.deinit();</code>
4	<code>const allocator = gpa.allocator();</code>
5	
6	<code>const stdout = std.io.getStdOut().writer();</code>
7	
8	<code>var gestor = GestorTareas.init();</code>
9	<code>try gestor.cargar();</code>
10	
11	<code>var args = try std.process.argsWithAllocator(allocator);</code>
12	<code>defer args.deinit();</code>
13	
14	<code>_ = args.next(); // Saltar nombre del programa</code>
15	
16	<code>const comando = args.next() orelse {</code>

17	try mostrar_ayuda(stdout);
18	return;
19	};
20	
21	if (std.mem.eql(u8, comando, "agregar")) {
22	const titulo = args.next() orelse {
23	try stdout.print("Error: falta el titulo\n", .{});
24	return;
25	};
26	const id = try gestor.agregar(titulo, .media);
27	try gestor.guardar();
28	try stdout.print("Tarea #{id} agregada\n", .{id});
29	} else if (std.mem.eql(u8, comando, "listar")) {
30	try gestor.listar(.todas, stdout);
31	} else if (std.mem.eql(u8, comando, "completar")) {
32	const id_str = args.next() orelse {
33	try stdout.print("Error: falta el ID\n", .{});
34	return;
35	};
36	const id = std.fmt.parseInt(u32, id_str, 10) catch {
37	try stdout.print("Error: ID invalido\n", .{});
38	return;
39	};
40	gestor.completar(id) catch {
41	try stdout.print("Error: tarea no encontrada\n", .{});
42	return;
43	};

44	<code>try gestor.guardar();</code>
45	<code>try stdout.print("Tarea #{id} completada\n", .{id});</code>
46	<code>} else {</code>
47	<code>try mostrar_ayuda(stdout);</code>
48	<code>}</code>
49	<code>}</code>
50	
51	<code>fn mostrar_ayuda(writer: anytype) !void {</code>
52	<code>try writer.print(</code>
53	<code>\\Gestor de Tareas - Comandos:</code>
54	<code>\\ agregar <titulo> - Agrega una nueva tarea</code>
55	<code>\\ listar - Muestra todas las tareas</code>
56	<code>\\ completar <id> - Marca una tarea como completada</code>
57	<code>\\</code>
58	<code>, .{id});</code>
59	<code>}</code>



Consejo

Observa el patrón de gestión de memoria en main: inicializamos el allocator al principio y usamos defer para su limpieza. Esto asegura que no haya fugas de memoria, sin importar en que punto termine el programa. Es el patrón estándar para aplicaciones CLI en Zig.

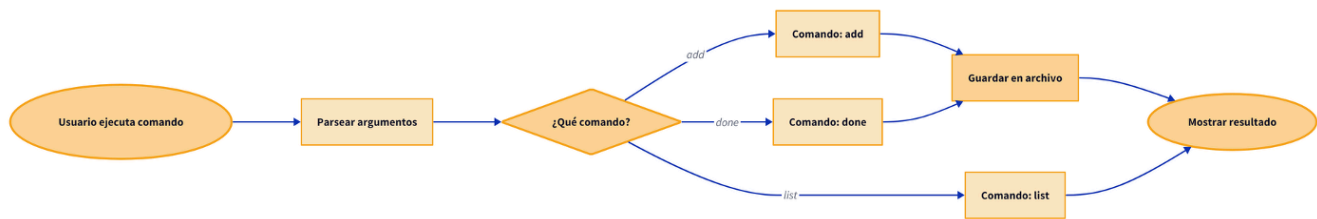


Figura 25 - Flujo de comandos

Veamos cómo se usa el programa en la práctica. Después de compilar con 'zig build-exe tareas.zig', puedes ejecutar los siguientes comandos:

1	\$./tareas agregar "Estudiar Zig"
2	Tarea #1 agregada
3	
4	\$./tareas agregar "Hacer ejercicio"
5	Tarea #2 agregada
6	
7	\$./tareas listar
8	[] * #1: Estudiar Zig
9	[] * #2: Hacer ejercicio
10	
11	\$./tareas completar 1
12	Tarea #1 completada
13	
14	\$./tareas listar
15	[x] * #1: Estudiar Zig
16	[] * #2: Hacer ejercicio

Las tareas se guardan automáticamente en `tareas.txt`, así que, si cierras el programa y lo vuelves a abrir, tus tareas seguirán ahí.

Este proyecto base puede extenderse de muchas formas. Podrías agregar la capacidad de cambiar la prioridad de una tarea, implementar fechas de vencimiento, agregar categorías o etiquetas, o crear un modo interactivo que muestre un menú en lugar de usar argumentos de línea de comandos.

Otra extensión interesante sería implementar un formato de archivo más sofisticado, como JSON, para almacenar las tareas. Esto requeriría un parser pero haría los datos más legibles y compatibles con otras herramientas



Consejo

Un buen ejercicio es agregar estas características uno por uno. Cada extensión te dará practica adicional y profundizara tu comprensión de Zig.

Observa como este proyecto relativamente pequeño utiliza casi todo lo que has aprendido en el libro. Las estructuras y enums modelan los datos. Los métodos encapsulan la lógica. El manejo de errores hace el programa robusto. Las operaciones de E/S permiten persistencia y comunicación con el usuario.

También nota las decisiones de diseño orientadas a Zig: usar arrays de tamaño fijo para evitar allocators donde no son necesarios, retornar errores en lugar de lanzar excepciones, y usar defer para garantizar limpieza de recursos.

Y con esto llegamos al final de este libro. Has recorrido un largo camino desde ese primer "Hola Mundo" hasta construir una aplicación completa. Zig es un lenguaje que recompensa la curiosidad y la práctica. Mientras más lo uses, más apreciaras su diseño cuidadoso y sus decisiones bien fundamentadas.

No pienses en este libro como el final de tu aprendizaje, sino como el comienzo. La documentación oficial de Zig, los ejemplos en el repositorio de la librería estándar, y los proyectos de código abierto son excelentes recursos para seguir aprendiendo.

Te animo a que tomes el proyecto de este capítulo y lo hagas tuyo. Agrégale características, experimenta con diferentes enfoques, y no tengas miedo de cometer errores. Cada error es una oportunidad de aprendizaje. El compilador de Zig está diseñado para darte mensajes de error claros que te guíen hacia la solución.

Gracias por acompañarme en este viaje a través de Zig. Espero que este libro te haya dado las herramientas y la confianza para emprender tus propios proyectos. ¡Feliz programación!

EJERCICIOS

Pon a prueba tus conocimientos con los siguientes ejercicios

AGREGAR PRIORIDAD AL COMANDO

Modifica el comando 'agregar' para que acepte un parámetro opcional de prioridad: 'agregar "Titulo" alta'. Si no se especifica, usa prioridad media.

Ejercicio

COMANDO ELIMINAR

Implementa un comando 'eliminar <id>' que elimine una tarea por su ID. Asegúrate de manejar el caso donde el ID no existe.

Ejercicio

GLOSARIO

Variable Espacio de almacenamiento con un nombre simbolico cuyo valor puede cambiar. En Zig se declara con 'var'.

Slice Vista o referencia a una porcion contigua de un array. Tiene un puntero al inicio y una longitud (len).

Binario Sistema numérico base 2 (0 y 1) utilizado por las computadoras. También se refiere al archivo ejecutable resultante de la compilación.

Bloque Conjunto de instrucciones agrupadas entre llaves {}. Define un ámbito (scope) para las variables.

Booleano Tipo de dato que solo puede tomar dos valores: verdadero (true) o falso (false).

Buffer Espacio de memoria temporal utilizado para almacenar datos mientras se mueven de un lugar a otro.

Bug Error o fallo en un programa que produce resultados inesperados o comportamientos incorrectos.

Casting Proceso de convertir un valor de un tipo de dato a otro (ej. de i32 a f64).

Comptime Palabra clave en Zig que indica que el código debe ejecutarse o resolverse en tiempo de compilación, no en tiempo de ejecución.

Concatenar Unir dos o más cadenas de texto o colecciones en una sola.

Constante Valor que se define una vez y no puede cambiar durante la ejecución del programa. En Zig se declara con 'const'.

Defer Palabra clave en Zig que programa una instrucción para ser ejecutada justo antes de que el bloque actual termine.

Enumeración (Enum) Tipo de dato definido por el usuario que consiste en un conjunto de constantes con nombre.

Error Unión Tipo especial en Zig (ej. !u32) que puede contener un valor válido o un error.

Escape Sequence Combinación de caracteres (como \n) que representa un carácter especial dentro de un string.

Función Bloque de código reutilizable que recibe parámetros y puede devolver un valor.

Heap (Montículo) Area de memoria para almacenamiento dinámico. Los datos aquí persisten hasta que se liberan explícitamente.

Identificador Nombre dado a una variable, función, constante o tipo definido por el usuario.

Inferencia de Tipos Capacidad del compilador para deducir automáticamente el tipo de una variable basándose en su valor inicial.

Iteración Repetición de un conjunto de instrucciones, comúnmente mediante bucles (for, while).

Librería Estándar Conjunto de módulos y funciones proporcionados por el lenguaje (en Zig, 'std') para realizar tareas comunes.

Método Función asociada a un struct o tipo, invocada con sintaxis de punto (obj.método()).

Módulo Unidad de código (generalmente un archivo) que puede ser importada y utilizada por otros módulos.

Null Valor especial que representa la ausencia de un valor. En Zig, se maneja mediante tipos opcionales (?T).

Optional Tipo que puede contener un valor o null; se denota como ?T en Zig.

Overflow Error que ocurre cuando el resultado de una operación aritmética excede la capacidad de almacenamiento del tipo de dato.

Padding Bytes de relleno agregados a una estructura para asegurar que los campos estén alineados en memoria.

Puntero Variable que almacena la dirección de memoria de otro valor.

Recursión Técnica donde una función se llama a si misma para resolver un problema.

Scope (Ambito) Región del programa donde una variable o símbolo es visible y puede usarse.

Sentinel Valor especial utilizado para marcar el final de una secuencia de datos (ej. el 0 al final de strings en C).

Slice Vista o referencia a una porción contigua de un array. Tiene un puntero al inicio y una longitud (len).

Stack (Pila) Area de memoria para almacenamiento temporal de variables locales y llamadas a funciones. Es rápida y se limpia automáticamente.

String Literal Secuencia de caracteres escrita directamente en el código, encerrada entre comillas.

Struct Estructura de datos compuesta que agrupa variables de diferentes tipos bajo un mismo nombre.

Switch Estructura de control que permite ejecutar diferentes bloques de código basándose en el valor de una variable.

Tag Unión Tipo de dato que puede almacenar uno de varios tipos diferentes, similar a una unión en C pero con seguridad de tipos.

Test Bloque de prueba automatizada en Zig, definido con la palabra clave test.

Undefined Valor especial en Zig que indica que una variable no tiene un valor inicial definido (usar con precaución).

Variable Espacio de almacenamiento con un nombre simbólico cuyo valor puede cambiar. En Zig se declara con 'var'.

Void Tipo que indica la ausencia de valor. Usado comúnmente como tipo de retorno de funciones que no devuelven nada.

While Estructura de control de flujo que repite un bloque de código mientras una condición sea verdadera.

BIBLIOGRAFÍA

Bryant, R. E., & O'Halláron, D. R. (2015). Computer Systems: A Programmer's Perspective (3.^a ed.). Pearson.

Cro, L. (2019, 5 de agosto). What is Zig's Comptime? Loris Cro's Blog. <https://kristoff.it/blog/what-is-zig-comptime/>

Cro, L. (2020, 28 de septiembre). Zig's New Relationship with LLVM. Loris Cro's Blog. <https://kristoff.it/blog/zig-new-relationship-llvm/>

Hinson-Hasty, G. (2024). Systems Programming with Zig. Manning Publications.

Kelley, A. (2016, 8 de febrero). Introduction to the Zig Programming Language. Andrew Kelley's Blog. <https://andrewkelley.me/post/intro-to-zig.html>

Kelley, A. (2017, 21 de enero). Practical Data Oriented Design. Andrew Kelley's Blog. <https://andrewkelley.me/post/practical-data-oriented-design.html>

Kelley, A. (2018, 24 de enero). Unsafe Zig is Safer Than Unsafe Rust. Andrew Kelley's Blog. <https://andrewkelley.me/post/unsafe-zig-safer-than-unsafe-rust.html>

Kernighan, B. W., & Ritchie, D. M. (1988). The C Programming Language (2.^a ed.). Prentice Hall.

Mali, G., & Gupta, S. (2024). Pragma driven shared memory parallelism in Zig by supporting OpenMP loop directives. En 2024 IEEE International Conference on High Performance Computing. IEEE.

Nystrom, R. (2021). Crafting Interpreters. Genever Benning.

Seguin, K. (2021). Learning Zig. Recuperado de https://openmymind.net/learning_zig/

Stepanov, A., & McJones, P. (2009). Elements of Programming. Addison-Wesley.

Zig Software Foundation. (2024a). Zig Language Reference. <https://ziglang.org/documentation/master/>

Zig Software Foundation. (2024b). Zig Standard Library Documentation. <https://ziglang.org/documentation/master/std/>

Zig Software Foundation. (2024c). Zig Build System. <https://ziglang.org/learn/build-system/>

Ziglings Project. (2023). Ziglings: Learn Zig by fixing tiny broken programs. GitHub. <https://github.com/ratfactor/ziglings>

