GNU Compiler Collection (gcc)

Pasos de Compilación, Visualización Intermedia y Optimización

Del Preprocesamiento al Código Ejecutable usando GCC

Resumen: Esta guía tiene como objetivo principal enseñar a los estudiantes de ingeniería a desglosar el proceso completo de compilación con GCC, desde el código fuente en C hasta el ejecutable final. A través de ejemplos prácticos reproducibles, mediciones cuantitativas y visualizaciones, el lector aprenderá a inspeccionar cada etapa intermedia (preprocesado, ensamblador, objeto), comprender el impacto real de diferentes niveles de optimización (-O0 a -O3, -Os) en el tamaño del código y el tiempo de ejecución, y aplicar mejores prácticas para proyectos universitarios y profesionales.

Curso: Sistemas Operativos

German Sánchez Torres, Ph.D.

Versión: 1.0

Índice general

Índice de figuras

Índice de cuadros

Capítulo 1

Introducción

1.1 Objetivos de la Guía

Esta guía tiene como propósito fundamental proporcionar a los estudiantes de ingeniería en sistemas y afines a la computación una comprensión profunda y práctica del proceso de compilación utilizando el GNU Compiler Collection (GCC). Los objetivos específicos son:

1. **Entender el flujo completo de compilación de GCC:** Desde el código fuente en C hasta el archivo ejecutable final, comprendiendo cada etapa del proceso y su importancia en el desarrollo de software.
2. **Aprender a inspeccionar etapas intermedias:** Dominar las técnicas para visualizar y analizar el código preprocesado, el ensamblador generado y los archivos objeto, desarrollando habilidades de depuración a bajo nivel.
3. **Analizar el impacto de optimizaciones:** Evaluar cuantitativamente cómo diferentes niveles de optimización (-O0, -O1, -O2, -O3, -Os) afectan el tamaño del código binario y el tiempo de ejecución, permitiendo tomar decisiones informadas en proyectos reales.

1.2 Público Objetivo

Esta guía está diseñada específicamente para:

- **Estudiantes de 3er semestre o superior** de Ingeniería en Sistemas Computacionales, Ingeniería en Computación o carreras afines.
- **Desarrolladores junior** que desean profundizar en el entendimiento de las herramientas de compilación.
- **Entusiastas de la programación de bajo nivel** interesados en comprender la relación entre código C y ensamblador.
- **Investigadores académicos** que necesitan analizar el rendimiento de algoritmos a nivel de código máquina.

Aunque la guía está orientada a estudiantes de nivel intermedio, los conceptos se presentan de manera progresiva, permitiendo que incluso estudiantes de semestres iniciales puedan seguir el material con dedicación.

1.3 Prerrequisitos

Para aprovechar al máximo esta guía, se recomienda contar con los siguientes conocimientos previos:

1.3.1 Conocimientos de Programación

- **Lenguaje C:** Familiaridad con sintaxis básica, estructuras de control (if, for, while), funciones, punteros y manejo básico de memoria.
- **Conceptos de algoritmos:** Comprensión de complejidad temporal y espacial básica.
- **Programación estructurada:** Capacidad para escribir y leer código modular.

1.3.2 Entorno de Trabajo

- **Terminal/Linux:** Conocimientos básicos de navegación por directorios, ejecución de comandos y redirección de salida.
- **Editor de texto:** Capacidad para editar archivos de código (recomendado: VS Code, Vim, o Nano).
- **Git (opcional):** Para clonar repositorios de ejemplos.

1.3.3 Hardware y Software Mínimo

Tabla 1.1: Requisitos mínimos del sistema

Componente	Requisito
Sistema Operativo	Linux (Ubuntu 20.04+), WSL2, o macOS
Memoria RAM	4 GB mínimo (8 GB recomendado)
Espacio en disco	2 GB libres
Procesador	x86_64 (Intel/AMD) o ARM64
GCC	Versión 9.0 o superior

1.4 Metodología

Esta guía adopta un enfoque **práctico y experimental**, fundamentado en los siguientes principios pedagógicos:

1.4.1 Aprendizaje Basado en Ejemplos

Cada concepto teórico se acompaña de ejemplos de código reproducibles. El lector puede ejecutar todos los comandos y programas presentados en su propio entorno.

1.4.2 Progresión Estructurada

El contenido se organiza de lo simple a lo complejo:

1. **Fundamentos:** Comprensión básica de GCC y sus componentes
2. **Descomposición:** Análisis detallado de cada paso de compilación
3. **Optimización:** Experimentación con diferentes niveles de optimización
4. **Herramientas avanzadas:** Uso de depuradores y perfiladores
5. **Proyectos prácticos:** Aplicación integrada de conocimientos

1.5 Estructura General de la Guía

Esta guía se organiza en siete capítulos principales, más anexos con soluciones y referencias:

Capítulo 1: Introducción
Contexto, objetivos y metodología de la guía (este capítulo).

Capítulo 2: Fundamentos de GCC
Historia, instalación, componentes y opciones básicas del compilador.

Capítulo 3: Los Pasos de Compilación
Desglose detallado del flujo completo: preprocesamiento, generación de ensamblador, ensamblado y enlace.

Capítulo 4: Niveles de Optimización
Análisis exhaustivo de -O0, -O1, -O2, -O3, -Os, con experimentos cuantitativos y visualizaciones.

Capítulo 5: Herramientas Avanzadas
Depuración con GDB, profiling con perf, y otras utilidades profesionales.

Capítulo 6: Ejercicios y Prácticas
Problemas graduados de dificultad básica, intermedia y avanzada.

Capítulo 7: Conclusión
Resumen de aprendizajes, mejores prácticas y limitaciones de GCC.

Se recomienda seguir los capítulos en orden secuencial, ya que cada uno construye sobre los conceptos del anterior. Sin embargo, los capítulos 5 y 6 pueden consultarse de manera independiente como referencia.

1.6 Recursos Adicionales

Para complementar el aprendizaje, se recomiendan los siguientes recursos:

1.6.1 Documentación Oficial

- **GCC Online Documentation:** https://gcc.gnu.org/onlinedocs/
- **GNU Binutils:** https://sourceware.org/binutils/docs/
- **GDB Manual:** https://sourceware.org/gdb/current/onlinedocs/gdb/

1.6.2 Herramientas Online

- **Compiler Explorer (godbolt.org):** Visualización interactiva de ensamblador generado por múltiples compiladores.
- **Godbolt Compiler Explorer:** https://godbolt.org

1.6.3 Libros Recomendados

- *Computer Systems: A Programmer's Perspective* (Bryant & O'Hallaron)
- *The Art of Assembly Language* (Randall Hyde)
- *Linkers and Loaders* (John R. Levine)

Los enlaces a recursos externos pueden cambiar. Si un enlace no funciona, se recomienda buscar el recurso por nombre en un motor de búsqueda.

Capítulo 2

Fundamentos de GCC

2.1 ¿Qué es GCC?

El **GNU Compiler Collection (GCC)** es un conjunto de compiladores de código abierto desarrollado por el Proyecto GNU. Originalmente diseñado como compilador de C (GNU C Compiler), hoy en día GCC soporta múltiples lenguajes de programación incluyendo C++, Objective-C, Fortran, Ada, Go y D.

2.1.1 Historia de GCC

GCC fue creado en **1987** por Richard Stallman como parte del proyecto GNU, con el objetivo de proporcionar un compilador libre para el sistema operativo GNU. La primera versión pública, GCC 1.0, fue lanzada ese mismo año y solo soportaba el lenguaje C.

Tabla 2.1: Evolución histórica de GCC

Año	Versión	Características principales
1987	1.0	Compilador C inicial
1992	2.0	Soporte para C++
1997	2.95	Mejoras significativas en optimización
2001	3.0	Nueva arquitectura interna, soporte para más lenguajes
2005	4.0	Optimizaciones a nivel de árbol (tree-ssa)
2012	4.7	Soporte para C++11
2015	5.0	Mejoras en diagnósticos, soporte C++14
2019	9.0	Soporte C++2a, mejoras en OpenMP
2020	10.0	Soporte para C++20, COBOL
2021	11.0	Mejoras en C++20, soporte para C++23
2022	12.0	Optimizaciones de vectorización mejoradas
2023	13.0	Mejoras en C++23, nuevas optimizaciones

2.1.2 Componentes de GCC

GCC no es un único programa, sino una **colección de herramientas** que trabajan en conjunto:

`gcc` Compilador principal de C. Es el driver que coordina todas las etapas de compilación.

`g++` Compilador de C++. Incluye soporte para la biblioteca estándar de C++ y enlaza automáticamente con libstdc++.

`gfortran`
Compilador de Fortran moderno (Fortran 95, 2003, 2008, 2018).

`gnat` Compilador de Ada, parte del proyecto GCC.

`go` Compilador del lenguaje Go (gccgo).

`cpp` El preprocesador de C, aunque generalmente se invoca automáticamente.

`as` El ensamblador de GNU (GNU Assembler), parte de binutils.

`ld` El enlazador de GNU (GNU Linker), también parte de binutils.

Aunque cada lenguaje tiene su propio front-end (analizador sintáctico y semántico), todos comparten el mismo **back-end** de optimización y generación de código, lo que garantiza consistencia en la calidad del código generado.

2.1.3 Versiones Recomendadas

Para seguir esta guía, se recomienda utilizar **GCC 9.0 o superior**. Las versiones más recientes (GCC 13+) ofrecen:

- Mejores diagnósticos de errores y advertencias
- Optimizaciones más agresivas y efectivas
- Soporte para estándares modernos de C (C11, C17, C23)
- Mejor integración con herramientas de análisis estático

2.2 Instalación y Configuración

2.2.1 Instalación en Linux

Ubuntu/Debian

```
# Actualizar repositorios
sudo apt update

# Instalar GCC, G++ y herramientas esenciales
sudo apt install build-essential gcc g++ gdb

# Instalar herramientas adicionales de desarrollo
sudo apt install binutils valgrind perf-tools-unstable
```

Código 2.1: Instalación en Ubuntu/Debian

Fedora/RHEL/CentOS

```
# Usando dnf (Fedora)
sudo dnf install gcc gcc-c++ gdb make

# Usando yum (RHEL/CentOS)
sudo yum install gcc gcc-c++ gdb make

# Grupo de herramientas de desarrollo
sudo dnf groupinstall "Development Tools"
```

Código 2.2: Instalación en Fedora/RHEL

Arch Linux

```
# Instalar base-devel que incluye GCC
sudo pacman -S base-devel gdb
```

Código 2.3: Instalación en Arch Linux

2.2.2 Instalación en macOS

En macOS, GCC se instala generalmente a través de **Homebrew**, ya que el compilador por defecto de Xcode (clang) es diferente.

```
# Instalar Homebrew (si no esta instalado)
/bin/bash -c "$(curl -fsSL
    https://raw.githubusercontent.com/Homebrew/install/HEAD/install.sh)"

# Instalar GCC
brew install gcc

# Verificar instalacion
gcc --version
```

Código 2.4: Instalación en macOS

En macOS, el comando `gcc` puede apuntar a Clang en lugar de GCC genuino. Para usar GCC instalado por Homebrew, utilice `gcc-13` o la versión específica instalada.

2.2.3 Instalación en Windows

Opción 1: WSL2 (Recomendada)

Windows Subsystem for Linux 2 proporciona un entorno Linux completo:

```
# En PowerShell (como administrador)
wsl --install
```

```
# Una vez instalado, abrir WSL y seguir instrucciones de Linux
sudo apt update
sudo apt install build-essential gcc g++ gdb
```

Código 2.5: Instalación en WSL2

Opción 2: MinGW-w64

```
# Descargar e instalar MSYS2 desde https://www.msys2.org/

# En terminal MSYS2
pacman -S mingw-w64-x86_64-gcc mingw-w64-x86_64-gdb
```

Código 2.6: Instalacion con MSYS2

2.2.4 Verificación de la Instalación

Después de la instalación, verifique que GCC esté correctamente configurado:

```
# Verificar version de GCC
gcc --version

# Salida esperada (ejemplo):
# gcc (Ubuntu 13.2.0-...) 13.2.0
# Copyright (C) 2023 Free Software Foundation, Inc.

# Verificar ubicacion del compilador
which gcc

# Verificar herramientas relacionadas
as --version # Ensamblador
ld --version # Enlazador
gdb --version # Depurador
```

Código 2.7: Verificacion de GCC

2.3 Opciones Básicas de Compilación

2.3.1 Compilación Simple

Una vez que hemos escrito el programa en un archivo programa.c, la forma más básica de compilarlo es C:

```
# Compilar y generar ejecutable 'programa'
gcc -o programa programa.c

# Ejecutar
./programa
```

Código 2.8: Compilacion basica

2.3.2 Flags Comunes

Tabla 2.2: Opciones comunes de GCC

Flag	Descripción
`-o archivo`	Especifica el nombre del archivo de salida
`-Wall`	Habilita todas las advertencias comunes (**W**arn **all**)
`-Wextra`	Habilita advertencias adicionales
`-Werror`	Trata las advertencias como errores
`-g`	Incluye información de depuración
`-std=c99`	Usa el estándar C99
`-std=c11`	Usa el estándar C11
`-std=c17`	Usa el estándar C17 (C18)
`-O0`	Sin optimización (por defecto)
`-O1`	Optimización básica
`-O2`	Optimización agresiva (recomendada)
`-O3`	Optimización muy agresiva
`-Os`	Optimización para tamaño
`-c`	Solo compila a objeto (.o), no enlaza
`-S`	Genera código ensamblador (.s)
`-E`	Solo preprocesa
`-v`	Modo verboso (muestra comandos internos)
`-save-temps`	Guarda archivos intermedios

2.3.3 Ejemplo Inicial: Hola Mundo

En esta sección construiremos el clásico *Hola Mundo* para establecer una base común: un archivo fuente mínimo, una salida verificable y un flujo de compilación controlado. Aunque el programa es trivial, sirve para entender qué produce GCC en cada etapa y qué artefactos intermedios aparecen durante el proceso.

Creemos nuestro primer programa para explorar el proceso de compilación paso a paso.

```c
#include <stdio.h>

int main(void) {
    printf("Hola, Mundo!\\n");
    return 0;
}
```

Código 2.9: programa.c - Hola Mundo

Compilación Paso a Paso

La compilación no ocurre en "un solo paso": GCC ejecuta internamente varias fases (preprocesamiento, generación de ensamblador, ensamblado y enlace). Al separarlas explícitamente, podemos inspeccionar los archivos intermedios (`.i`, `.s`, `.o`) y comprender mejor dónde surgen errores, advertencias o impactos de optimización.

```
# Paso 1: Preprocesamiento
```

```
gcc -E programa.c -o programa.i

# Paso 2: Generacion de ensamblador
gcc -S programa.i -o programa.s

# Paso 3: Ensamblado (genera codigo objeto)
gcc -c programa.s -o programa.o

# Paso 4: Enlace (genera ejecutable)
gcc programa.o -o programa

# Ejecutar
./programa
```

Código 2.10: Compilacion paso a paso

En la práctica, estos cuatro pasos se pueden realizar con un único comando: `gcc -o programa programa.c`. Sin embargo, comprender cada etapa por separado es fundamental para el debugging y la optimización avanzada.

2.3.4 Compilación con Opciones Recomendadas

En C, compilar "bien" no es solo generar un ejecutable: también implica activar advertencias útiles, habilitar símbolos de depuración y fijar explícitamente el estándar del lenguaje. Estas opciones ayudan a detectar errores temprano (por ejemplo, conversiones peligrosas o uso de variables no inicializadas), facilitan el análisis con herramientas de debugging, y hacen el comportamiento del compilador más predecible entre máquinas. A continuación se muestran combinaciones prácticas para desarrollo y producción, junto con distintos niveles de optimización.

Para desarrollo, se recomienda usar:

```
# Desarrollo (con advertencias y debug)
gcc -Wall -Wextra -g -std=c11 -o programa programa.c

# Produccion (con optimizacion)
gcc -Wall -O2 -std=c11 -o programa programa.c

# Maxima optimizacion (cuidado con comportamiento indefinido)
gcc -Wall -O3 -std=c11 -o programa programa.c
```

Código 2.11: Opciones recomendadas para desarrollo

2.4 Estructura de un Proyecto C

Aunque un programa pequeño puede compilarse desde un único archivo, la mayoría de proyectos reales crecen rápidamente y requieren organización. Separar el código fuente, los encabezados y los artefactos generados (objetos y ejecutables) mejora la mantenibilidad, evita dependencias circulares y facilita la colaboración. La siguiente estructura es una convención

común: permite compilar por módulos, reutilizar componentes y automatizar el build sin mezclar archivos generados con el código del repositorio.

Un proyecto C típico tiene la siguiente estructura:

```
mi_proyecto/
|-- src/ # Codigo fuente
| |-- main.c
| |-- utils.c
| |-- algo.c
|-- include/ # Headers publicos
| |-- utils.h
| |-- algo.h
|-- build/ # Archivos objeto (generado)
|-- bin/ # Ejecutables (generado)
|-- Makefile # Script de compilacion
|-- README.md # Documentacion
```

Código 2.12: Estructura tipica de proyecto C

2.4.1 Compilación de Proyectos con Múltiples Archivos

Cuando el proyecto se divide en varios archivos, el proceso típico es: (1) compilar cada `.c` a un archivo objeto `.o` y (2) enlazar todos los objetos para generar el ejecutable final. Este enfoque permite recompilar solo los módulos que cambian y reduce tiempos de build.

```
# Compilar cada fuente a objeto
gcc -c -I include src/utils.c -o build/utils.o
gcc -c -I include src/algo.c -o build/algo.o
gcc -c -I include src/main.c -o build/main.o

# Enlazar todos los objetos
gcc build/utils.o build/algo.o build/main.o -o bin/programa
```

Código 2.13: Compilacion de proyecto multi-archivo

Para proyectos grandes, el uso de **Make** o **CMake** es esencial para automatizar el proceso de compilación y solo recompilar los archivos modificados.

2.5 Resumen del Capítulo

En este capítulo hemos cubierto:

- La historia y evolución de GCC como proyecto de software libre
- Los componentes principales de la colección de compiladores
- Procedimientos de instalación en diferentes sistemas operativos
- Opciones básicas de compilación y flags comunes
- El proceso de compilación paso a paso con un ejemplo “Hola Mundo”

- Estructura recomendada para proyectos C

En el próximo capítulo, exploraremos en detalle cada uno de los pasos de compilación, aprendiendo a visualizar y analizar las etapas intermedias del proceso.

Capítulo 3

Los Pasos de Compilación

Este capítulo constituye el **núcleo fundamental** de la guía. Analizaremos en profundidad cada etapa del proceso de compilación, desde el código fuente en C hasta el ejecutable final.

3.1 Diagrama General del Flujo

El proceso de compilación con GCC consta de **cuatro etapas principales**:

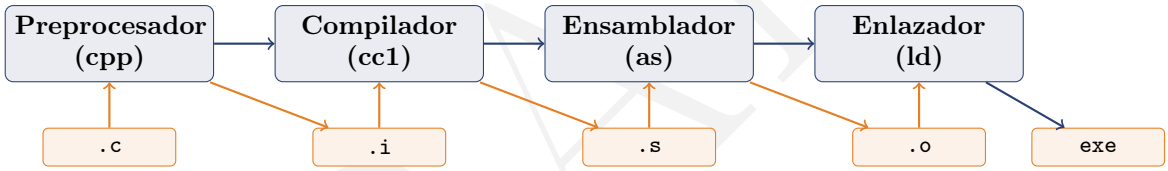


Tabla 3.1: Resumen de etapas de compilación

Etapa	Ext.	Comando	Herramienta	Propósito
Preproceso	.i	-E	cpp	Expansión textual
Compilación	.s	-S	cc1	C a ensamblador
Ensamblado	.o	-c	as	Ensamblador a binario
Enlace	—	—	ld	Enlace de objetos

3.2 Preprocesamiento

3.2.1 Descripción del Proceso

El **preprocesador** (`cpp`) es la primera etapa del proceso de compilación. Su función es realizar transformaciones **pura y exclusivamente textuales** sobre el código fuente:

- **Expansión de macros:** Reemplaza las definiciones `#define` por su valor.
- **Inclusión de headers:** Inserta el contenido de archivos `#include`.
- **Eliminación de comentarios:** Remueve comentarios `/* */` y `//`.
- **Compilación condicional:** Evalúa directivas `#if`, `#ifdef`, `#ifndef`.

- **Directivas de línea:** Inserta información para debugging con `#line`.

3.2.2 Comando para Visualizar

Algunos ejemplos para visualizar el resultado. La descripción de las opciones está en la Tabla 2.2.

```
# Preprocesar y guardar salida
gcc -E programa.c > programa.i

# Especificar nombre de salida explicitamente
gcc -E programa.c -o programa.i

# Con opciones adicionales (ver macros definidas)
gcc -E -dM programa.c | head -50
```

Código 3.1: Generar codigo preprocesado

3.2.3 Ejemplo Práctico

Consideremos el siguiente programa con múltiples elementos preprocesables:

```
1  #include <stdio.h>
2
3  #define PI 3.14159
4  #define CUADRADO(x) ((x) * (x))
5  #define DEBUG
6
7  int main(void) {
8      double radio = 5.0;
9      double area = PI * CUADRADO(radio);
10
11     #ifdef DEBUG
12     printf("[DEBUG] Radio: %f\\n", radio);
13     printf("[DEBUG] Area calculada\\n");
14     #endif
15
16     printf("Area del circulo: %f\\n", area);
17     return 0;
18 }
```

Código 3.2: ejemplo_prepro.c - Codigo con macros

Resultado del Preprocesamiento

En este ejemplo se puede visualizar el reemplazo de la macro CUADRADO(x) ((x) * (x)) sobre la llamada de la línea 9.

```
# Codigo original (lineas relevantes)
#define PI 3.14159
area = PI * CUADRADO(radio);

# Codigo preprocesado
```

```
gcc -E ejemplo_prepro.c | grep -A2 "area ="
# area = 3.14159 * ((radio) * (radio));
```

Código 3.3: Comparacion de codigo original vs preprocesado

El archivo `.i` resultante contiene:

- Todo el contenido de `stdio.h` (puede ser miles de líneas)
- Las macros expandidas
- Sin comentarios
- Código condicional resuelto

3.2.4 Directivas del Preprocesador

Tabla 3.2: Directivas comunes del preprocesador

Directiva	Descripción y Uso
`#define`	Define macros constantes o funcionales
`#undef`	Elimina una definición previa
`#include`	Incluye contenido de otro archivo
`#ifdef`	Compila si el identificador está definido
`#ifndef`	Compila si el identificador NO está definido
`#if`	Compila si la expresión es verdadera
`#elif`	"Else if" para compilación condicional
`#else`	Alternativa en compilación condicional
`#endif`	Cierra bloque condicional
`#error`	Genera un error de compilación
`#warning`	Genera una advertencia
`#pragma`	Directiva específica del compilador
`#line`	Establece número de línea y archivo

Práctica 3.1: Inspección de Macros

1. Cree un archivo con macros anidadas complejas.
2. Use `gcc -E` para ver el resultado expandido.
3. Use `gcc -E -dM` para listar todas las macros predefinidas.
4. Compare el número de líneas entre `.c` y `.i` (explore el comando linux wc -l).

3.3 Generación de Código Ensamblador

3.3.1 Descripción del Proceso

El compilador propiamente dicho (`cc1` para C) traduce el código preprocesado a **código ensamblador** específico de la arquitectura objetivo. Esta es la etapa más compleja del proceso:

- **Análisis léxico:** Verifica que el texto contenga *símbolos y palabras válidas* (identificadores, literales, operadores, palabras reservadas) y los agrupa en unidades significativas. **Input:** código preprocesado (`.i`). **Output:** secuencia de *tokens*.
- **Análisis sintáctico:** Comprueba que la secuencia de tokens siga la *gramática del lenguaje* y construye la estructura jerárquica del programa. **Input:** tokens. **Output:** árbol de sintaxis abstracta (AST).
- **Análisis semántico:** Valida el *significado* del programa: tipos, declaraciones previas, compatibilidad de operaciones, conversiones, alcances (scope) y reglas del estándar. **Input:** AST + tabla de símbolos. **Output:** AST anotado + diagnósticos (errores/advertencias).
- **Optimización:** Aplica transformaciones para mejorar *tiempo, tamaño o consumo* sin cambiar el comportamiento definido por el estándar (según el nivel `-O*`). **Input:** representación intermedia (IR) derivada del AST. **Output:** IR optimizada.
- **Generación de código:** Traduce la IR optimizada a instrucciones de la arquitectura objetivo, asigna registros y emite el ensamblador final. **Input:** IR optimizada + modelo de máquina (target). **Output:** archivo ensamblador (`.s`).

3.3.2 Comando para Visualizar

```
# Generar ensamblador (sintaxis AT&T por defecto)
gcc -S programa.c

# Generar ensamblador con sintaxis Intel
gcc -S -masm=intel programa.c

# Con informacion adicional de depuracion
gcc -S -fverbose-asm programa.c

# Especificar nombre de salida
gcc -S programa.c -o programa.s
```

Código 3.4: Generar codigo ensamblador

3.3.3 Sintaxis ATT vs Intel

En x86, la diferencia entre AT&T e Intel no es "otro ensamblador", sino *dos notaciones* para escribir las mismas instrucciones. GCC (vía GAS) usa AT&T por defecto: los *registros* llevan prefijo `%`, los *inmediatos* llevan `$`, el *orden* suele ser *origen* → *destino*, y el *tamaño* puede indicarse con sufijos en la instrucción (p. ej., `movb/movw/movl/movq`). La sintaxis Intel, activable en GCC con `-masm=intel`, es familiar para muchos programadores porque escribe *destino, luego origen*, no usa `%`/`$`, y representa memoria con corchetes (p. ej., `[ebp+8]`). Ambas describen el mismo código máquina, solo cambia la forma de escribirlo.

GCC usa por defecto la sintaxis ATT, pero la sintaxis Intel puede ser mas familiar para algunos programadores:

Tabla 3.3: Comparacion de sintaxis ATT vs Intel

Caracteristica	**ATT**	**Intel**
Orden de operandos	movl \$5, %eax	mov eax, 5
Registros	%eax	eax
Inmediatos	\$5	5
Direccionamiento	8(%ebp)	[ebp + 8]

3.3.4 Ejemplo: Función con Bucle

```c
int suma_arreglo(int *arr, int n) {
    int suma = 0;
    for (int i = 0; i < n; i++) {
        suma += arr[i];
    }
    return suma;
}
```

Código 3.5: bucle.c - Funcion con bucle for

Código Ensamblador Generado (sintaxis Intel)

```
# gcc -S -masm=intel -O0 bucle.c

suma_arreglo:
    push rbp
    mov rbp, rsp
    mov QWORD PTR [rbp-24], rdi # arr
    mov DWORD PTR [rbp-28], esi # n
    mov DWORD PTR [rbp-4], 0 # suma = 0
    mov DWORD PTR [rbp-8], 0 # i = 0
    jmp .L2
.L3:
    mov eax, DWORD PTR [rbp-8] # i
    cdqe
    lea rdx, [0+rax*4]
    mov rax, QWORD PTR [rbp-24] # arr
    add rax, rdx
    mov eax, DWORD PTR [rax] # arr[i]
    add DWORD PTR [rbp-4], eax # suma += arr[i]
    add DWORD PTR [rbp-8], 1 # i++
.L2:
    mov eax, DWORD PTR [rbp-8] # i
    cmp eax, DWORD PTR [rbp-28] # i < n
    jl .L3
    mov eax, DWORD PTR [rbp-4] # return suma
    pop rbp
    ret
```

Código 3.6: bucle.s - Codigo ensamblador

3.3.5 Registros x86-64 Comunes

Tabla 3.4: Registros x86-64 y sus usos

64-bit	32-bit	Uso típico
RAX	EAX	Valor de retorno, acumulador
RBX	EBX	Registro base (preservado)
RCX	ECX	Contador de bucles
RDX	EDX	Datos, I/O
RSI	ESI	Índice fuente
RDI	EDI	Índice destino, 1er argumento
RBP	EBP	Puntero de base de pila
RSP	ESP	Puntero de cima de pila
R8-R15	R8D-R15D	Registros adicionales (x86-64)

En la convención de llamada System V AMD64 ABI (Linux/macOS), los primeros 6 argumentos enteros se pasan en registros: `RDI`, `RSI`, `RDX`, `RCX`, `R8`, `R9`.

3.3.6 Desensamblado con objdump

Para ver el código máquina junto con el ensamblador:

```
# Compilar a objeto
gcc -c bucle.c -o bucle.o

# Desensamblar mostrando codigo maquina y ensamblador
objdump -d -M intel bucle.o

# Desensamblado con informacion de secciones
objdump -h bucle.o
```

Código 3.7: Desensamblado con objdump

Práctica 3.2: Análisis de Ensamblador

1. Compile una función simple con `-O0`, `-O2` y `-O3`.
2. Compare el código ensamblador generado en cada caso.
3. Identifique las optimizaciones aplicadas.
4. Use `objdump -d` para ver el código máquina real.

3.4 Ensamblado y Enlace

3.4.1 Ensamblado: De .s a .o

El **ensamblador** (`as`) traduce el código ensamblador legible por humanos a **código máquina binario** en formato objeto:

```
# Ensamblar archivo .s a .o
as programa.s -o programa.o

# O usando gcc (que invoca as internamente)
gcc -c programa.s -o programa.o

# Directamente desde C (combina compilacion y ensamblado)
gcc -c programa.c -o programa.o
```

Código 3.8: Ensamblado de codigo

Contenido de un Archivo Objeto

Un archivo `.o` contiene:

- **Código máquina:** Instrucciones binarias ejecutables
- **Tabla de símbolos:** Funciones y variables definidas/referenciadas
- **Secciones:** `.text` (código), `.data` (datos inicializados), `.bss` (datos no inicializados)
- **Información de relocación:** Direcciones que deben ajustarse durante el enlace

3.4.2 Visualización de Símbolos con nm

```
# Listar simbolos de un archivo objeto
nm programa.o

# Salida tipica:
# 0000000000000000 T main
# U printf
# U puts

# Simbolos ordenados por direccion
nm -n programa.o

# Simbolos con tamano
nm -S programa.o
```

Código 3.9: Inspeccion de simbolos con nm

Tabla 3.5: Tipos de símbolos en nm

Letra	Significado
`T`	Símbolo en sección de texto (código)
`D`	Símbolo en sección de datos inicializados
`B`	Símbolo en sección BSS (datos no inicializados)
`U`	Símbolo no definido (externo)
`R`	Símbolo en sección de sólo lectura
`W`	Símbolo débil

3.4.3 Enlace: Generación del Ejecutable

El **enlazador** (`ld`) combina múltiples archivos objeto y bibliotecas en un ejecutable final:

```
# Enlazar un archivo objeto
gcc programa.o -o programa

# Enlazar multiples objetos
gcc main.o utils.o algo.o -o programa

# Enlazar con bibliotecas
gcc main.o -o programa -lm -lpthread
```

Código 3.10: Enlace de archivos objeto

Funciones del Enlazador

1. **Resolución de símbolos:** Conecta referencias con definiciones
2. **Relocación:** Ajusta direcciones de memoria
3. **Fusión de secciones:** Combina secciones similares
4. **Inclusión de bibliotecas:** Agrega código de bibliotecas

3.4.4 Dependencias con ldd

```
# Listar bibliotecas compartidas requeridas
ldd programa

# Salida tipica:
# linux-vdso.so.1
# libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6
# /lib64/ld-linux-x86-64.so.2
```

Código 3.11: Ver dependencias de bibliotecas

3.4.5 Tabla Comparativa de Etapas

Tabla 3.6: Resumen completo de etapas de compilación

Etapa	Ext.	Comando	Herramienta	Propósito
Preproceso	.i	-E	cpp	Expansión de macros, inclusión de headers
Compilación	.s	-S	cc1	C a ensamblador (análisis + optimización)
Ensamblado	.o	-c	as	Ensamblador a código máquina
Enlace	—	—	ld	Combinación de objetos y bibliotecas

3.4.6 Comando de Un Solo Paso

```
# Todas las etapas en un comando
gcc -o programa programa.c

# Con todas las opciones recomendadas
gcc -Wall -Wextra -O2 -std=c11 -o programa programa.c

# Guardar archivos intermedios
gcc -save-temps -o programa programa.c
# Genera: programa.i, programa.s, programa.o, programa
```

Código 3.12: Compilacion completa en un paso

Use `-save-temps` durante el aprendizaje para conservar todos los archivos intermedios y poder inspeccionar cada etapa del proceso de compilación.

3.5 Resumen del Capítulo

En este capítulo hemos analizado en profundidad:

- El flujo completo de compilación en cuatro etapas
- El preprocesador: expansión de macros, inclusión de headers, compilación condicional
- La generación de código ensamblador: análisis y traducción
- El ensamblado: de código legible a binario objeto
- El enlace: combinación de objetos en ejecutables
- Herramientas de inspección: `nm`, `objdump`, `ldd`

En el próximo capítulo, exploraremos los diferentes niveles de optimización y su impacto cuantitativo en el rendimiento.

Capítulo 4

Niveles de Optimización

Este capítulo constituye la sección **más analítica** de la guía. Exploraremos los diferentes niveles de optimización de GCC, sus trade-offs y realizaremos experimentos cuantitativos para medir su impacto real.

4.1 Conceptos Teóricos

Al compilar con `-O1/-O2/-O3`, GCC no "acelera por magia": aplica transformaciones sobre una representación intermedia del programa para reducir instrucciones, mejorar el uso de caché y registros, y explotar capacidades del CPU (por ejemplo SIMD). El conjunto exacto de optimizaciones depende del nivel de optimización, de la arquitectura objetivo y de si el compilador puede asumir comportamiento definido (por eso el *undefined behavior* en C es crítico).

Optimización de instrucciones:
Reescribe secuencias para usar instrucciones equivalentes pero más baratas (latencia/throughput), combina operaciones, y elimina movimientos redundantes. Ej.: reemplazar `mul` por desplazamientos cuando el multiplicador es potencia de dos.

Optimización de bucles:
Transforma bucles porque suelen dominar el tiempo de ejecución. Incluye *unrolling* (desenrollado) para reducir saltos, *loop invariant code motion* para sacar cálculos constantes fuera del bucle, e *interchange* para mejorar localidad de memoria (accesos más contiguos).

Inlining:
Sustituye una llamada a función por el cuerpo de la función. Reduce el overhead de llamada y abre oportunidades para otras optimizaciones (propagación de constantes, eliminación de ramas). A cambio, puede aumentar el tamaño del binario.

Vectorización:
Detecta operaciones repetitivas sobre arreglos (p. ej., sumas elemento a elemento) y genera instrucciones SIMD (SSE/AVX/AVX-512) para procesar varios elementos por instrucción. Depende de alineación, dependencias y del `-march`.

Dead code elimination:
Elimina código que no tiene efectos observables: cálculos cuyo resultado no se usa, ramas inalcanzables, variables temporales redundantes. Suele activarse tras otras pasadas que vuelven evidente que algo "no aporta".

Constant folding:
Evalúa expresiones constantes en compilación (p. ej., `3*8` → `24`) y puede simplificar condiciones (`if (0)` → eliminado), reduciendo trabajo en tiempo de ejecución.

Register allocation:
Decide qué valores viven en registros y cuáles "derraman" a la pila. Un buen asignador reduce accesos a memoria, pero está limitado por la cantidad de registros y por el *lifetime* de cada variable.

Tail call optimization:
En llamadas de cola, reemplaza la llamada recursiva final por un salto, reutilizando el mismo frame de pila. Reduce consumo de stack y evita overhead en ciertos patrones recursivos (cuando el ABI y las condiciones lo permiten).

4.1.1 Trade-offs: Velocidad vs. Tamaño vs. Debuggabilidad

La optimización implica **compromisos** que deben considerarse:

Tabla 4.1: Trade-offs de optimización

Factor	Optimización Alta	Optimización Baja
Velocidad	Más rápido	Más lento
Tamaño	Generalmente mayor (-O3) o menor (-Os)	Mayor (-O0)
Tiempo compilación	Más lento	Más rápido
Debuggabilidad	Difícil (código reordenado)	Fácil (mapeo 1:1)
Uso memoria	Puede variar	Predecible
Consumo energía	Generalmente menor	Mayor

La optimización agresiva (-O3) puede cambiar el comportamiento de programas que dependen de comportamiento indefinido (undefined behavior) o que hacen suposiciones sobre el orden de evaluación.

4.2 Niveles de Optimización

4.2.1 Tabla Detallada de Niveles

Tabla 4.2: Niveles de optimización de GCC

Nivel	Flag	Enfoque	Tamaño	Velocidad	Uso recomendado
Ninguno	-O0	Sin optimizaciones	Máximo	Lento	Desarrollo, debugging
Básico	-O1	Optimizaciones simples	Reducido	Medio	Compilación rápida
Medio	-O2	Análisis global	Muy reducido	Rápido	**Producción (recomendado)**
Avanzado	-O3	Vectorización, inlining	Variable	Muy rápido	Código CPU-intensivo
Tamaño	-Os	Reducción de código	Mínimo	Medio-Rápido	Sistemas embebidos
Rápido	-Ofast	Agresivo (no estándar)	Variable	Máximo	Cuando se aceptan trade-offs

4.2.2 Descripción de Cada Nivel

-O0 (Sin Optimización)

- Compilación más rápida
- Código más fácil de depurar (mapeo directo fuente-ensamblador)
- Útil durante desarrollo activo
- Incluye toda la información de depuración

-O1 (Optimización Básica)

- Dead code elimination
- Constant propagation
- Simplificación de expresiones
- Sin optimizaciones que afecten significativamente el tiempo de compilación

-O2 (Optimización Estándar)

- Todas las optimizaciones de -O1
- Function inlining (moderado)
- Loop optimizations
- Instruction scheduling
- **Recomendado para la mayoría de aplicaciones de producción**

-O3 (Optimización Agresiva)

- Todas las optimizaciones de -O2
- Vectorización automática (SIMD)
- Aggressive inlining
- Loop vectorization
- Puede aumentar significativamente el tamaño del código

-Os (Optimización para Tamaño)

- Todas las optimizaciones de -O2 que no aumentan tamaño
- Preferencia por código más compacto
- Ideal para sistemas embebidos con memoria limitada

-Ofast (Máxima Velocidad)

- Todas las optimizaciones de -O3
- Ignora estándares estrictos (puede romper código válido)
- Usar con precaución

4.3 Métodos de Medición

4.3.1 Medición de Tamaño del Código

Comando size

```
# Compilar con diferentes optimizaciones
gcc -O0 -o programa_O0 programa.c
gcc -O2 -o programa_O2 programa.c
gcc -O3 -o programa_O3 programa.c

# Medir tamano de secciones
size programa_O0 programa_O2 programa_O3
```

Código 4.1: Medicion de tamano con size

Comando ls -lh

```
# Tamano del archivo completo
ls -lh programa_*
```

Código 4.2: Tamano total del ejecutable

4.3.2 Medición de Tiempo de Ejecución

Método Simple: time

```
# Usar time del sistema (no el built-in de shell)
/usr/bin/time -f "Tiempo real: %e s" ./programa

# Formato detallado
/usr/bin/time -v ./programa
```

Código 4.3: Medicion basica con time

Método Preciso: clock_gettime()

```c
#include <stdio.h>
#include <time.h>

// Funcion a benchmark (ejemplo: fibonacci)
long long fibonacci(int n) {
    if (n <= 1) return n;
    return fibonacci(n - 1) + fibonacci(n - 2);
}

int main(void) {
    struct timespec inicio, fin;

    // Obtener tiempo inicial
    clock_gettime(CLOCK_MONOTONIC, &inicio);

    // Ejecutar funcion a medir
    long long resultado = fibonacci(40);

    // Obtener tiempo final
    clock_gettime(CLOCK_MONOTONIC, &fin);

    // Calcular diferencia en nanosegundos
    long segundos = fin.tv_sec - inicio.tv_sec;
    long nanosegundos = fin.tv_nsec - inicio.tv_nsec;
    double tiempo_total = segundos + nanosegundos * 1e-9;

    printf("Resultado: %lld\\n", resultado);
    printf("Tiempo: %.6f segundos\\n", tiempo_total);

    return 0;
}
```

Código 4.4: benchmark.c - Medicion precisa

Script de Benchmark Automatizado

```
#!/bin/bash

# Niveles de optimizacion a probar
```

```
NIVELES="O0 O1 O2 O3 Os"

# Numero de ejecuciones por prueba
ITERACIONES=10

echo "Compilando programas..."
for nivel in $NIVELES; do
    gcc -$nivel -o programa_$nivel benchmark.c -lm
done

echo "Ejecutando benchmarks..."
echo "Nivel,Tiempo_Promedio,Tamano"
for nivel in $NIVELES; do
    total=0
    for i in $(seq 1 $ITERACIONES); do
        tiempo=$(/usr/bin/time -f "%e" ./programa_$nivel 2>&1 | tail -1)
        total=$(echo "$total + $tiempo" | bc)
    done
    promedio=$(echo "scale=6; $total / $ITERACIONES" | bc)
    tamano=$(stat -c%s programa_$nivel)
    echo "$nivel,$promedio,$tamano"
done
```

Código 4.5: benchmark.sh - Script de benchmark

4.4 Experimentos Prácticos

4.4.1 Programa Benchmark: Multiplicación de Matrices

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define N 300

void multiplicar_matrices(int A[N][N], int B[N][N], int C[N][N]) {
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            C[i][j] = 0;
            for (int k = 0; k < N; k++) {
                C[i][j] += A[i][k] * B[k][j];
            }
        }
    }
}

int main(void) {
    static int A[N][N], B[N][N], C[N][N];

    // Inicializar matrices
    for (int i = 0; i < N; i++)
        for (int j = 0; j < N; j++) {
            A[i][j] = rand() % 100;
            B[i][j] = rand() % 100;
        }
```

```
27
28      struct timespec inicio, fin;
29      clock_gettime(CLOCK_MONOTONIC, &inicio);
30
31      multiplicar_matrices(A, B, C);
32
33      clock_gettime(CLOCK_MONOTONIC, &fin);
34
35      double tiempo = (fin.tv_sec - inicio.tv_sec) +
36                      (fin.tv_nsec - inicio.tv_nsec) * 1e-9;
37
38      printf("N=%d, Tiempo: %.4f segundos\\n", N, tiempo);
39      printf("Elemento de prueba: C[0][0] = %d\\n", C[0][0]);
40
41      return 0;
42  }
```

Código 4.6: matmul.c - Multiplicacion de matrices

4.4.2 Resultados Esperados

Tabla 4.3: Resultados típicos de optimización (matmul.c, N=300)

Nivel	Tiempo (s)	Tamaño (KB)	Speedup
-O0	2.500	25.0	1.00x
-O1	0.850	22.5	2.94x
-O2	0.180	21.0	13.89x
-O3	0.045	23.5	55.56x
-Os	0.420	18.5	5.95x

Los resultados exactos varían significativamente según el hardware, la versión de GCC y la naturaleza del algoritmo. Los valores mostrados son representativos para ilustrar las tendencias.

4.4.3 Visualización de Resultados

Gráfico de Barras: Tiempo vs Nivel

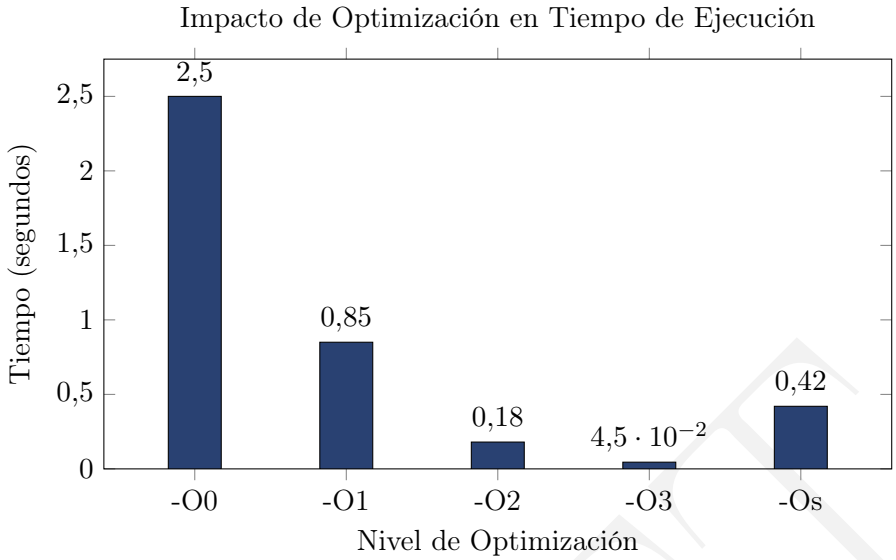
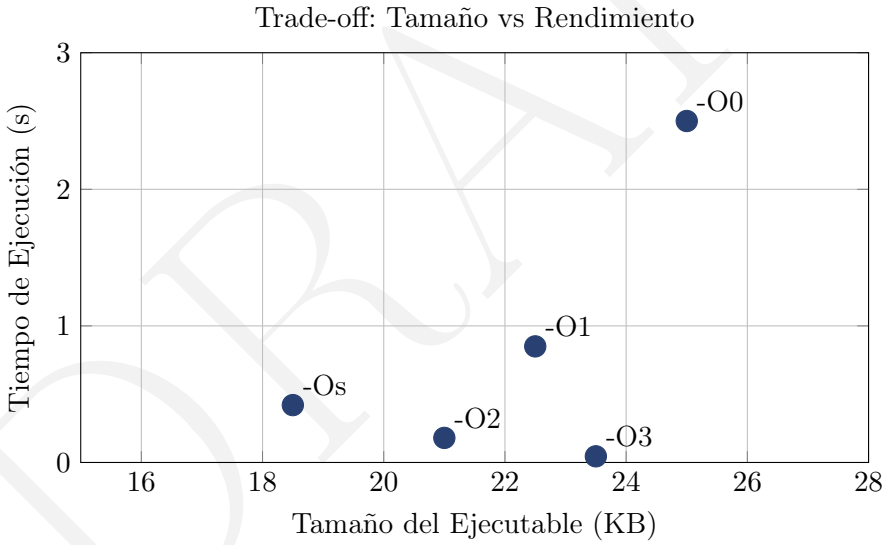


Gráfico de Dispersión: Tamaño vs Tiempo



4.5 Análisis de Resultados

4.5.1 ¿Cuándo Usar Cada Nivel?

-O0:
Durante el desarrollo activo y debugging. El mapeo directo entre código fuente y ensamblador facilita enormemente la depuración.

-O1:
Cuando se necesita un balance entre tiempo de compilación y rendimiento básico. Útil para proyectos grandes en desarrollo.

-O2:
Opción recomendada para producción. Ofrece excelente rendimiento sin los riesgos de -O3.

-O3:
Para código intensivo en CPU donde cada ciclo cuenta: simulaciones científicas, procesamiento de imágenes, machine learning.

-Os: Sistemas embebidos, dispositivos IoT, o cualquier entorno con memoria limitada.

-Ofast:
Solo cuando se comprenden completamente los trade-offs y se acepta el comportamiento no estándar.

4.5.2 Riesgos de Optimizaciones Agresivas

Comportamiento Indefinido

```c
#include <stdio.h>

int main(void) {
    int x = 1;
    int y = x + 1; // Optimizado: y = 2

    // Si x se modifica de forma no visible al compilador
    // -O3 puede asumir que y sigue siendo 2

    printf("%d\\n", y);
    return 0;
}
```

Código 4.7: ejemplo_ub.c - Comportamiento indefinido

Flags de Seguridad

```
# Desactivar optimizaciones que asumen comportamiento definido
gcc -O3 -fwrapv -fno-strict-aliasing -o programa programa.c

# -fwrapv: Aritmetica con wrap-around definido
# -fno-strict-aliasing: Desactiva optimizaciones de aliasing
```

Código 4.8: Flags para comportamiento seguro

4.5.3 Comparación con Clang

Tabla 4.4: Comparación GCC vs Clang

Aspecto	**GCC**	**Clang**
Velocidad compilación	Más lenta	Más rápida
Diagnósticos	Buenos	Excelentes
Optimización -O2	Similar	Similar
Optimización -O3	Muy agresiva	Muy agresiva
Soporte C++20	Completo	Completo
Licencia	GPL	Apache 2.0

```
# Clang usa los mismos flags de optimizacion
clang -O2 -o programa programa.c
clang -O3 -march=native -o programa programa.c
```

Código 4.9: Compilar con Clang

4.6 Resumen del Capítulo

En este capítulo hemos cubierto:

- Los conceptos teóricos detrás de la optimización de compiladores
- Los seis niveles de optimización de GCC (-O0, -O1, -O2, -O3, -Os, -Ofast)
- Métodos para medir tamaño y tiempo de ejecución
- Experimentos prácticos con benchmarks reales
- Visualización de resultados con gráficos
- Análisis de trade-offs y cuándo usar cada nivel
- Riesgos de optimizaciones agresivas y cómo mitigarlos
- Comparación breve con Clang

La regla de oro: Comience con -O2 para producción. Use -O3 solo después de verificar que no introduce problemas. Use -O0 para debugging.