



UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO



# Compiladores

## Práctica 0 Sistema de procesamiento de Lenguaje

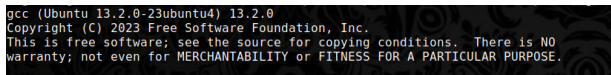
Sarah Sophía Olivares García

318360638

## 1. Introducción

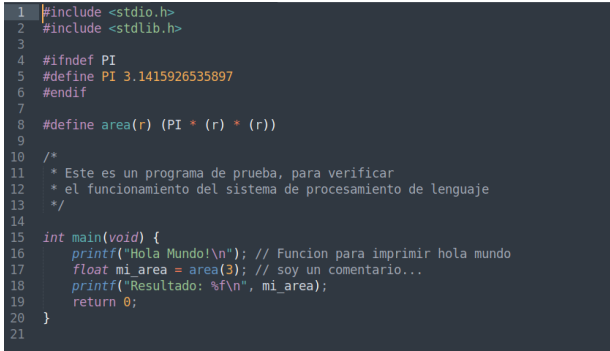
En esta práctica se busca en la primera parte realizar el preprocesamiento del archivo C, examinado el contenido de `stdio.h`, comparado ambos archivos, y entendiendo el proceso realizado por el preprocesador `cpp`.

## 2. Instalación y Escritura



```
gcc (Ubuntu 13.2.0-23ubuntu4) 13.2.0
Copyright (C) 2023 Free Software Foundation, Inc.
This is free software; see the source for copying conditions. There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

Figura 1: Programa.i compilado



```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 #ifndef PI
5 #define PI 3.1415926535897
6 #endif
7
8 #define area(r) (PI * (r) * (r))
9
10 /*
11  * Este es un programa de prueba, para verificar
12  * el funcionamiento del sistema de procesamiento de lenguaje
13  */
14
15 int main(void) {
16     printf("Hola Mundo!\n"); // Funcion para imprimir hola mundo
17     float mi_area = area(3); // soy un comentario...
18     printf("Resultado: %f\n", mi_area);
19     return 0;
20 }
21
```

Figura 2: Archivo del sistema operativo

### 3. cpp programa.c > programa.i

```
extern double erand48 (unsigned short int __xsubi[3]) __attribute__((__nothrow__, __leaf__)) __attribute__((__nonnull__(1)));

extern long int lrand48 (void) __attribute__((__nothrow__, __leaf__));
extern long int nrand48 (unsigned short int __xsubi[3])
    __attribute__((__nothrow__, __leaf__)) __attribute__((__nonnull__(1)));

extern long int mrand48 (void) __attribute__((__nothrow__, __leaf__));
extern long int jrand48 (unsigned short int __xsubi[3])
    __attribute__((__nothrow__, __leaf__)) __attribute__((__nonnull__(1)));

extern void srand48 (long int __seedval) __attribute__((__nothrow__, __leaf__));
extern unsigned short int *seed48 (unsigned short int __seed16v[3])
    __attribute__((__nothrow__, __leaf__)) __attribute__((__nonnull__(1)));
extern void lcong48 (unsigned short int __param[7]) __attribute__((__nothrow__, __leaf__)) __attribute__((__nonnull__(1)));

struct drand48_data
{
    unsigned short int __x[3];
    unsigned short int __old_x[3];
    unsigned short int __c;
    unsigned short int __init;
    __extension__ unsigned long long int __a;
};

extern int drand48_r (struct drand48_data * __restrict __buffer,
    double * __restrict __result) __attribute__((__nothrow__, __leaf__)) __attribute__((__nonnull__(1, 2)));
extern int erand48_r (unsigned short int __xsubi[3],
    struct drand48_data * __restrict __buffer,
    double * __restrict __result) __attribute__((__nothrow__, __leaf__)) __attribute__((__nonnull__(1, 2)));
```

Figura 3: Programa.i compilado

```
__nonnull__(1));
extern int fsetpos64 (FILE * __stream, const fpos64_t * __pos) __nonnull__(1);
#endif

/* Clear the error and EOF indicators for STREAM. */
extern void clearerr (FILE * __stream) __THROW __nonnull__(1);
/* Return the EOF indicator for STREAM. */
extern int feof (FILE * __stream) __THROW __wur __nonnull__(1);
/* Return the error indicator for STREAM. */
extern int ferror (FILE * __stream) __THROW __wur __nonnull__(1);

#ifdef __USE_MISC
/* Faster versions when locking is not required. */
extern void clearerr_unlocked (FILE * __stream) __THROW __nonnull__(1);
extern int feof_unlocked (FILE * __stream) __THROW __wur __nonnull__(1);
extern int ferror_unlocked (FILE * __stream) __THROW __wur __nonnull__(1);
#endif

/* Print a message describing the meaning of the value of errno.

   This function is a possible cancellation point and therefore not
   marked with __THROW. */
extern void perror (const char * __s) __COLD;

#ifdef __USE_POSIX
/* Return the system file descriptor for STREAM. */
extern int fileno (FILE * __stream) __THROW __wur __nonnull__(1);
#endif /* Use POSIX. */

#ifdef __USE_MISC
/* Faster version when locking is not required. */
extern int fileno_unlocked (FILE * __stream) __THROW __wur __nonnull__(1);
#endif

#ifdef __USE_POSIX2
/* Close a stream opened by popen and return the status of its child.
```

Figura 4: Archivo del sistema operativo

## (a) Localizar y examinar `stdio.h`

### Localización:

Para localizar el archivo `stdio.h` en el sistema, podemos usar el siguiente comando:

```
find /usr -name "stdio.h" 2>/dev/null
```

Este comando buscará el archivo `stdio.h` en los directorios del sistema, como `/usr/include`, donde normalmente se encuentra.

### Exámen del contenido:

Una vez localizado, podemos examinar su contenido con cualquier editor de texto o usando un comando como `cat` o `less`:

```
cat /usr/include/stdio.h
```

`stdio.h` es un archivo de cabecera de la biblioteca estándar de C que contiene definiciones de macros, constantes y declaraciones de funciones para la entrada/salida estándar como `printf`, `scanf`, `fgets`, etc.

## (b) Comparación de `programa.i` con `stdio.h`

### Contenido de `programa.i`:

El archivo `programa.i` contiene el código fuente original de `programa.c`, pero con todas las directivas del preprocesador ya expandidas. Esto incluye la expansión de macros, la inclusión de archivos de cabecera (como `stdio.h`), y la evaluación de cualquier código condicional.

### Similitudes más notables:

- **Inclusión de `stdio.h`:** Después de preprocesar, `programa.i` incluirá el contenido del archivo `stdio.h` donde hayamos utilizado la directiva `#include <stdio.h>` en `programa.c`.
- **Expansión de macros:** Las macros definidas en `stdio.h`, como `NULL` o `EOF`, estarán presentes en `programa.i`.
- **Estructura similar:** Aunque `programa.i` contiene más código debido a las expansiones, mantendrá la estructura lógica del archivo original, con el contenido de `stdio.h` insertado en las posiciones correspondientes.

## (c) Explicación de la ejecución de `cpp`

### Preprocesamiento:

Durante la ejecución del comando `cpp`, el preprocesador realiza las siguientes tareas:

- **Expansión de macros:** Reemplaza cualquier macro definida en el código con su valor correspondiente.
- **Inclusión de archivos de cabecera:** Inserta el contenido de los archivos de cabecera (como `stdio.h`) directamente en el código donde se haya especificado la directiva `#include`.
- **Evaluación de condicionales:** Procesa directivas condicionales (`#if`, `#ifdef`, `#endif`, etc.) y elimina o incluye bloques de código según las condiciones especificadas.
- **Eliminación de comentarios:** Elimina todos los comentarios del código fuente, ya que no son necesarios en las etapas posteriores de compilación.

## 4. Explicación de las opciones de compilación

### (a) Función de la opción `-Wall`

**Descripción:** La opción `-Wall` en `gcc` activa la mayoría de las advertencias del compilador. Estas advertencias te alertan sobre posibles problemas en tu código que, aunque no son errores, pueden llevar a comportamientos inesperados o fallos futuros.

**Importancia:** Al usar `-Wall`, podemos identificar y corregir problemas potenciales antes de que se conviertan en errores graves, mejorando la calidad del código.

### (b) Propósito de la opción `-S`

**Descripción:** La opción `-S` le indica a `gcc` que solo convierta el código fuente (en este caso, `programa.i`) en código ensamblador y no continúe con la creación de un archivo objeto o un ejecutable.

**Resultado:** El resultado es un archivo de ensamblador con extensión `.s`, que es una representación más baja del código en C, pero aún legible por humanos.

### (c) Contenido y extensión del archivo de salida

Contenido:

- Directivas de ensamblador para definir secciones y datos.
- Instrucciones en ensamblador que representan la función `main` del programa C.
- Constantes de solo lectura como cadenas de texto.
- Información sobre el compilador y datos de alineación.

**Extensión:** El archivo de salida tendrá la extensión `.s`. Si el nombre original del archivo era `programa.i`, el archivo ensamblador generado se llamará `programa.s`.

### (d) Programa invocado por `gcc` con la opción `-S`

**Descripción:** Cuando se utiliza la opción `-S`, `gcc` invoca el compilador C que traduce el código en C a código ensamblador. Este proceso es manejado internamente por la herramienta `cc1` (el compilador C interno de GCC), que realiza la traducción del código C al código ensamblador.

## 5. Resumen del Archivo Objeto (`.o`)

### (a) Contenido del Archivo `.o`

El archivo objeto (`programa.o`) es un archivo binario que contiene el código máquina correspondiente a las instrucciones en ensamblador de `programa.s`. Este código máquina es específico de la arquitectura del procesador en el que estás trabajando.

Además del código máquina, el archivo `.o` también contiene:

- **Tablas de símbolos:** Estas tablas relacionan los nombres de las funciones y variables con sus direcciones de memoria correspondientes.

- **Tablas de reubicación:** Estas tablas contienen información sobre las direcciones que deben ser modificadas cuando el archivo objeto se vincula con otros archivos objeto o bibliotecas para formar un ejecutable.
- **Datos de depuración:** Información adicional que permite a las herramientas de depuración rastrear el código fuente original durante la ejecución.

## (b) Descripción General del Contenido del Archivo .o y su Importancia

**Descripción general:** El archivo objeto .o es un fragmento del programa completo, que incluye el código en lenguaje máquina, tablas de símbolos y tablas de reubicación. No es un programa ejecutable por sí mismo, pero es un componente necesario en la fase de enlace, donde se combina con otros archivos objeto y bibliotecas para formar un ejecutable completo.

### Importancia:

- **Modularidad:** El archivo objeto permite que el código sea compilado por separado, facilitándole la división del trabajo en proyectos grandes.
- **Eficiencia en el desarrollo:** Al trabajar con archivos objeto, no necesitas recompilar todo el proyecto cuando cambias un solo archivo fuente, solo los archivos modificados.
- **Reutilización:** Las bibliotecas compartidas o estáticas se distribuyen en forma de archivos objeto, permitiendo su reutilización en múltiples programas.

## (c) Programa Invocado con el Comando as

**Programa invocado:** El comando **as** invoca el ensamblador de GNU (GNU Assembler), conocido como **as**. Este programa toma el código ensamblador (.s) y lo convierte en un archivo objeto (.o), que es un archivo binario con código máquina.

El ensamblador **as** es una parte fundamental de la cadena de herramientas de compilación de GCC, y es utilizado en casi todos los sistemas Unix/Linux para ensamblar código en múltiples arquitecturas de CPU.

## 6. Rutas para las dependencias

```
find /usr -name "crt1.o" 2>/dev/null
/usr/lib/x86_64-linux-gnu/crt1.o
```

```
find /usr -name "crti.o" 2>/dev/null
/usr/lib/x86_64-linux-gnu/crti.o
```

```
find /usr -name "crtbegin.o" 2>/dev/null
/usr/lib/gcc/x86_64-linux-gnu/11/crtbegin.o
/usr/lib/gcc/x86_64-linux-gnu/13/crtbegin.o
/usr/lib/gcc/x86_64-linux-gnu/9/crtbegin.o
/usr/lib/gcc/x86_64-linux-gnu/12/crtbegin.o
```

```
find /usr -name "crtend.o" 2>/dev/null
/usr/lib/gcc/x86_64-linux-gnu/11/crtend.o
```

```

/usr/lib/gcc/x86_64-linux-gnu/13/crtend.o
/usr/lib/gcc/x86_64-linux-gnu/9/crtend.o
/usr/lib/gcc/x86_64-linux-gnu/12/crtend.o

```

```

find /usr -name "crtn.o" 2>/dev/null
/usr/lib/x86_64-linux-gnu/crtn.o

```

## 7. Enlazar el Programa

Para enlazar el programa sin usar la opción PIE, ejecuta el siguiente comando en la terminal:

```

ld -dynamic-linker /lib64/ld-linux-x86-64.so.2 \
/usr/lib/x86_64-linux-gnu/crt1.o \
/usr/lib/x86_64-linux-gnu/crti.o \
/usr/lib/gcc/x86_64-linux-gnu/13/crtbegin.o \
-L/usr/lib/gcc/x86_64-linux-gnu/13 \
-L/usr/lib -L/lib -L/usr/lib \
programa.o \
-lgcc --as-needed -lgcc_s --no-as-needed -lc \
/usr/lib/gcc/x86_64-linux-gnu/13/crtend.o \
/usr/lib/x86_64-linux-gnu/crtn.o \
-o programa

```

Para enlazar el programa utilizando ‘gcc’, ejecuta el siguiente comando en la terminal:

```

gcc -o ejecutable programa.o

```

## 8. Sustituyendo las Rutas

Para enlazar el programa utilizando el segundo comando, reemplaza las rutas con las encontradas:

```

ld -o ejecutable -dynamic-linker /lib64/ld-linux-x86-64.so.2 -pie \
/usr/lib/x86_64-linux-gnu/crt1.o \
/usr/lib/x86_64-linux-gnu/crti.o \
/usr/lib/gcc/x86_64-linux-gnu/13/crtbegin.o \
programa.o \
-lc \
/usr/lib/gcc/x86_64-linux-gnu/13/crtend.o \
/usr/lib/x86_64-linux-gnu/crtn.o

```

## Descripción del Resultado del Enlace

Al ejecutar el comando de enlace proporcionado, se espera obtener los siguientes resultados:

### ■ Archivo Ejecutable:

- **Nombre del Archivo:** El archivo ejecutable generado se llamará `ejecutable` o `programa`, dependiendo del comando utilizado.
- **Ubicación:** El archivo ejecutable se guardará en el directorio donde se ejecutó el comando `ld`.

### ■ Proceso de Enlace:

- **Configuración de Archivos:** Los archivos de inicio (`crt1.o`, `crti.o`, `crtbegin.o`) y los archivos de finalización (`crtend.o`, `crtfn.o`) proporcionan el entorno y las funciones necesarias para la inicialización y finalización del programa.
- **Bibliotecas Enlazadas:** Las bibliotecas como `-lgcc`, `-lgcc_s`, y `-lc` se vinculan con el programa para proporcionar funciones estándar y de apoyo necesarias para la ejecución.
- **Generación del Ejecutable:** `ld` combinará el archivo objeto (`programa.o`) con los archivos y bibliotecas especificadas para producir un archivo ejecutable.

## 9. Ejecución de programa

- **Ejecutar el Programa:** Una vez generado el archivo ejecutable, podemos ejecutarlo en la terminal para ver los resultados de tu programa. Por ejemplo:

```
./programa
```

Este comando ejecutará el programa y, si como esta configurado correctamente, producir la salida esperada de:

A terminal window with a dark background. The first line shows the output "Hola Mundo!" in a light blue font. The second line shows "Resultado: 28.274334" in a light blue font. To the right of this, the number "283" is visible in a light blue font. Further to the right, the text "et ent" and "item" are partially visible in a light blue font.

## 10. Cambio en la Macro y Resultado Final

### (a) Generación del Archivo Preprocesado

Se ha modificado el archivo fuente 'programa.c' para quitar el comentario de la macro '#define PI'. El archivo modificado es el siguiente:

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

#define PI 3.1415926535897
```

```

#define area(r) (PI * (r) * (r))

/*
 * Este es un programa de prueba, para verificar
 * el funcionamiento del sistema de procesamiento de lenguaje
 */

int main(void) {
    printf("Hola Mundo!\n"); // Funcion para imprimir hola mundo
    float mi_area = area(3); // soy un comentario...
    printf("Resultado: %f\n", mi_area);
    return 0;
}

```

Para generar el archivo preprocesado, se utilizó el comando:

```
cpp programa.c > programa_nuevo.i
```

## (b) Cambios en la Ejecución Final

Antes de quitar el comentario de la macro PI, el valor de PI era 3.1416 si PI no estaba definido. Después de quitar el comentario y definir PI como 3.1415926535897, el valor utilizado en los cálculos de área cambia.

- **Valor del Área Antes del Cambio:**  $PI = 3.1416$

$$\text{Área} = 3.1416 \times 3^2 = 28.2744$$

- **Valor del Área Después del Cambio:**  $PI = 3.1415926535897$

$$\text{Área} = 3.1415926535897 \times 3^2 \approx 28.274333882308138$$

La diferencia en los resultados es mínima, pero observable dependiendo de la precisión mostrada por el formato `%f` en `printf`.

## 11. Explicación del Segundo Programa en C

A continuación se presenta una explicación de un segundo programa en lenguaje C que utiliza cuatro directivas del preprocesador distintas a las del primer programa.

### Explicación de las Directivas del Preprocesador

1. `#define MAX_SIZE 100`: Define una constante `MAX_SIZE` con el valor 100. Esta constante se utiliza para especificar el tamaño del array en el programa.



2. **#define SQUARE(x) ((x) \* (x))**: Define una macro que calcula el cuadrado de un número. La macro **SQUARE** se utiliza para inicializar los valores en el array del programa.
3. **#include config.h**: Incluye un archivo de encabezado **config.h**. Este archivo puede contener configuraciones adicionales o definiciones necesarias para el programa. En este caso, podría incluir definiciones adicionales que afectan el comportamiento del programa.
4. **#ifdef DEBUG ... #else ... #endif**: Utiliza una directiva condicional para incluir o excluir el código basado en si **DEBUG** está definido o no. Si **DEBUG** está definido, la macro **LOG** imprimirá mensajes de depuración. De lo contrario, **LOG** no hará nada.

## Archivo de Encabezado Ejemplo (config.h)

Para que el programa compile correctamente, se puede crear un archivo **config.h** con el siguiente contenido para activar la depuración:

```
#define DEBUG
```

## Instrucciones para Compilar y Ejecutar

1. **Guardar el archivo fuente** con un nombre como **programa2.c**.
2. **Crear el archivo de configuración config.h** con el contenido mencionado.
3. **Compilar el programa** utilizando el siguiente comando:

```
gcc -Wall programa2.c -o programa2
```

4. **Ejecutar el programa** con el siguiente comando:

```
./programa2
```

Este programa demuestra cómo se utilizan diversas directivas del preprocesador en C para configurar el comportamiento del código.