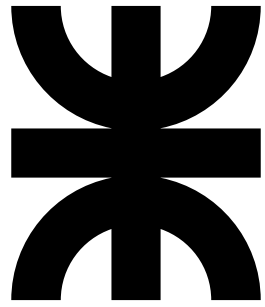
Informática II

El compilador de C del proyecto GNU (`gcc, g++`)

Gonzalo F. Perez Paina



Universidad Tecnológica Nacional
Facultad Regional Córdoba
UTN-FRC

– 2021 –

El compilador de C del proyecto GNU

Herramientas de compilación del proyecto GNU (GNU Compiler Collection):

El compilador de C del proyecto GNU

Herramientas de compilación del proyecto GNU (GNU Compiler Collection):

- Maneja varios dialectos de C: ANSI-C, tradicional (Kernighan & Ritchie), extensiones GNU, etc.

El compilador de C del proyecto GNU

Herramientas de compilación del proyecto GNU (GNU Compiler Collection):

- Maneja varios dialectos de C: ANSI-C, tradicional (Kernighan & Ritchie), extensiones GNU, etc.
- Puede compilar C++

El compilador de C del proyecto GNU

Herramientas de compilación del proyecto GNU (GNU Compiler Collection):

- Maneja varios dialectos de C: ANSI-C, tradicional (Kernighan & Ritchie), extensiones GNU, etc.
- Puede compilar C++
- Realiza la optimización del código

El compilador de C del proyecto GNU

Herramientas de compilación del proyecto GNU (GNU Compiler Collection):

- Maneja varios dialectos de C: ANSI-C, tradicional (Kernighan & Ritchie), extensiones GNU, etc.
- Puede compilar C++
- Realiza la optimización del código
- Genera información de depuración (debugging)

El compilador de C del proyecto GNU

Herramientas de compilación del proyecto GNU (GNU Compiler Collection):

- Maneja varios dialectos de C: ANSI-C, tradicional (Kernighan & Ritchie), extensiones GNU, etc.
- Puede compilar C++
- Realiza la optimización del código
- Genera información de depuración (debugging)
- Es un compilador cruzado (cross-compiler)
  `$ gcc --version`
  `$ gcc -v`
  (`--build`, `--host`, `--target`)

Compilando – Algunos ejemplos

hola.c

```c
#include <stdio.h>

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

Compilando – Algunos ejemplos

hola.c

```c
#include <stdio.h>

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

Compilación

```
> gcc hola.c
```

Compilando – Algunos ejemplos

hola.c

```c
#include <stdio.h>

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

Compilación

```
> gcc hola.c
```

(salida `a.out`)

Compilando – Algunos ejemplos

hola.c

```c
#include <stdio.h>

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

Compilación

```
> gcc hola.c
```

(salida `a.out`) Cómo se ejecuta?

Compilando – Algunos ejemplos

hola.c

```c
#include <stdio.h>

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

Compilación

```
> gcc hola.c
```

(salida `a.out`) Cómo se ejecuta?

Cómo cambiar el nombre a binario?

Compilando – Algunos ejemplos

hola.c

```c
#include <stdio.h>

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

Compilación

```
> gcc hola.c
```

(salida `a.out`) Cómo se ejecuta?

Cómo cambiar el nombre a binario?

```
> gcc hola.c -o hola
```

Compilando – Algunos ejemplos

hola.c

```c
#include <stdio.h>

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

Compilación

```
> gcc hola.c
```

(salida `a.out`) Cómo se ejecuta?

Cómo cambiar el nombre a binario?

```
> gcc hola.c -o hola
```

```
> gcc -Wall hola.c -o hola
```

Compilando – Algunos ejemplos

hola.c

```c
#include <stdio.h>

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

Compilación

```
> gcc hola.c
```

(salida `a.out`) Cómo se ejecuta?

Cómo cambiar el nombre a binario?

```
> gcc hola.c -o hola
```

```
> gcc -Wall hola.c -o hola
```

(habilita todas las advertencias)

Compilando – Algunos ejemplos

hola.c

```c
#include <stdio.h>

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

mal.c

```c
#include <stdio.h>

int main(void)
{
  printf("Dos y dos son %f\n", 4);
  return 0;
}
```

Compilación

```
> gcc hola.c
```

(salida `a.out`) Cómo se ejecuta?

Cómo cambiar el nombre a binario?

```
> gcc hola.c -o hola
```

```
> gcc -Wall hola.c -o hola
```

(habilita todas las advertencias)

Compilando – Algunos ejemplos

hola.c

```c
#include <stdio.h>

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

mal.c

```c
#include <stdio.h>

int main(void)
{
  printf("Dos y dos son %f\n", 4);
  return 0;
}
```

Compilación

```
> gcc hola.c
```

(salida `a.out`) Cómo se ejecuta?

Cómo cambiar el nombre a binario?

```
> gcc hola.c -o hola
```

```
> gcc -Wall hola.c -o hola
```

(habilita todas las advertencias)

Compilar y ejecutar

```
> gcc mal.c -o mal
> ./mal
Dos y dos son 0.000000
```

Compilando – Algunos ejemplos

hola.c

```c
#include <stdio.h>

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

mal.c

```c
#include <stdio.h>

int main(void)
{
  printf("Dos y dos son %f\n", 4);
  return 0;
}
```

Compilación

```
> gcc hola.c
```

(salida `a.out`) Cómo se ejecuta?

Cómo cambiar el nombre a binario?

```
> gcc hola.c -o hola
```

```
> gcc -Wall hola.c -o hola
```

(habilita todas las advertencias)

Compilar y ejecutar

```
> gcc mal.c -o mal
> ./mal
Dos y dos son 0.000000
```

Error de compilación (`-Wall`)

```
mal.c:5:10: warning: format '%f'
expects argument of type 'double
but argument 2 has type 'int'
[-Wformat=]
```

Compilando – Ejemplos con varios archivos fuentes

main.c

```c
#include "hola.h"

int main(void)
{
  hola("mundo");
  return 0;
}
```

Compilando – Ejemplos con varios archivos fuentes

main.c

```c
#include "hola.h"

int main(void)
{
  hola("mundo");
  return 0;
}
```

hola.c

```c
#include <stdio.h>
#include "hola.h"

void hola(const char * nombre)
{
  printf("Hola, %s!\n", nombre);
}
```

Compilando – Ejemplos con varios archivos fuentes

main.c

```c
#include "hola.h"

int main(void)
{
  hola("mundo");
  return 0;
}
```

hola.c

```c
#include <stdio.h>
#include "hola.h"

void hola(const char * nombre)
{
  printf("Hola, %s!\n", nombre);
}
```

hola.h

```c
void hola(const char * nombre);
```

Compilando – Ejemplos con varios archivos fuentes

main.c

```c
#include "hola.h"

int main(void)
{
  hola("mundo");
  return 0;
}
```

hola.c

```c
#include <stdio.h>
#include "hola.h"

void hola(const char * nombre)
{
  printf("Hola, %s!\n", nombre);
}
```

hola.h

```c
void hola(const char * nombre);
```

Compilación

```
> gcc -Wall main.c hola.c -o hola
```

Compilando – Ejemplos con varios archivos fuentes

main.c

```c
#include "hola.h"

int main(void)
{
  hola("mundo");
  return 0;
}
```

hola.c

```c
#include <stdio.h>
#include "hola.h"

void hola(const char * nombre)
{
  printf("Hola, %s!\n", nombre);
}
```

hola.h

```c
void hola(const char * nombre);
```

Compilación

```
> gcc -Wall main.c hola.c -o hola
```

Se puede compilar separadamente cada archivo fuente

```
> gcc -Wall -c main.c
> gcc -Wall -c hola.c
```

Compilando – Ejemplos con varios archivos fuentes

main.c

```c
#include "hola.h"

int main(void)
{
  hola("mundo");
  return 0;
}
```

hola.c

```c
#include <stdio.h>
#include "hola.h"

void hola(const char * nombre)
{
  printf("Hola, %s!\n", nombre);
}
```

hola.h

```c
void hola(const char * nombre);
```

Compilación

```
> gcc -Wall main.c hola.c -o hola
```

Se puede compilar separadamente cada archivo fuente

```
> gcc -Wall -c main.c
> gcc -Wall -c hola.c
```

y unirlo con el linker (enlazador)

```
> gcc main.o hola.o -o hola
```

Compilando – Ejemplos con varios archivos fuentes

main.c

```c
#include "hola.h"

int main(void)
{
  hola("mundo");
  return 0;
}
```

hola.c

```c
#include <stdio.h>
#include "hola.h"

void hola(const char * nombre)
{
  printf("Hola, %s!\n", nombre);
}
```

hola.h

```c
void hola(const char * nombre);
```

Compilación

```
> gcc -Wall main.c hola.c -o hola
```

Se puede compilar separadamente cada archivo fuente

```
> gcc -Wall -c main.c
> gcc -Wall -c hola.c
```

y unirlo con el linker (enlazador)

```
> gcc main.o hola.o -o hola
```

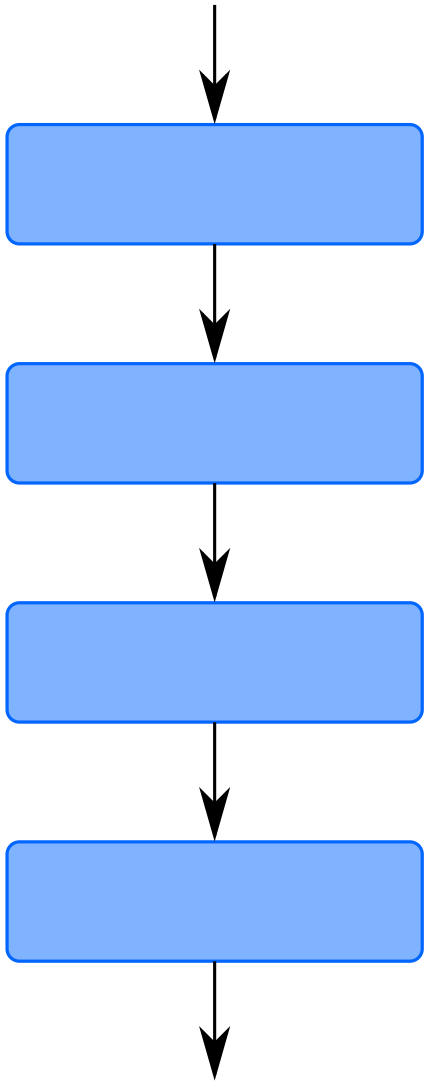
Esto permite modifica un archivo fuente y recompilar solo ese archivo.

Etapas de compilación/construcción

El proceso de compilación/construcción involucra 4 etapas (preprocesamiento, compilación, ensamblado, y enlazado).

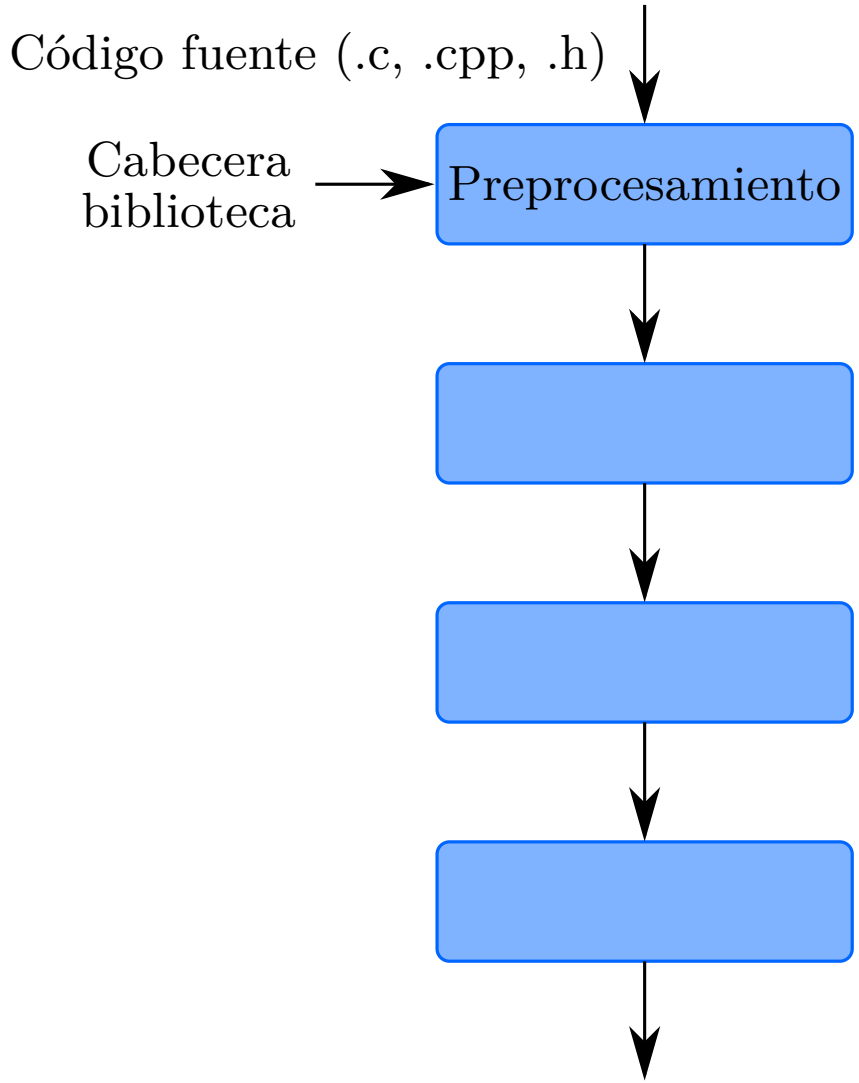
Etapas de compilación/construcción

El proceso de compilación/construcción involucra 4 etapas (preprocesamiento, compilación, ensamblado, y enlazado).



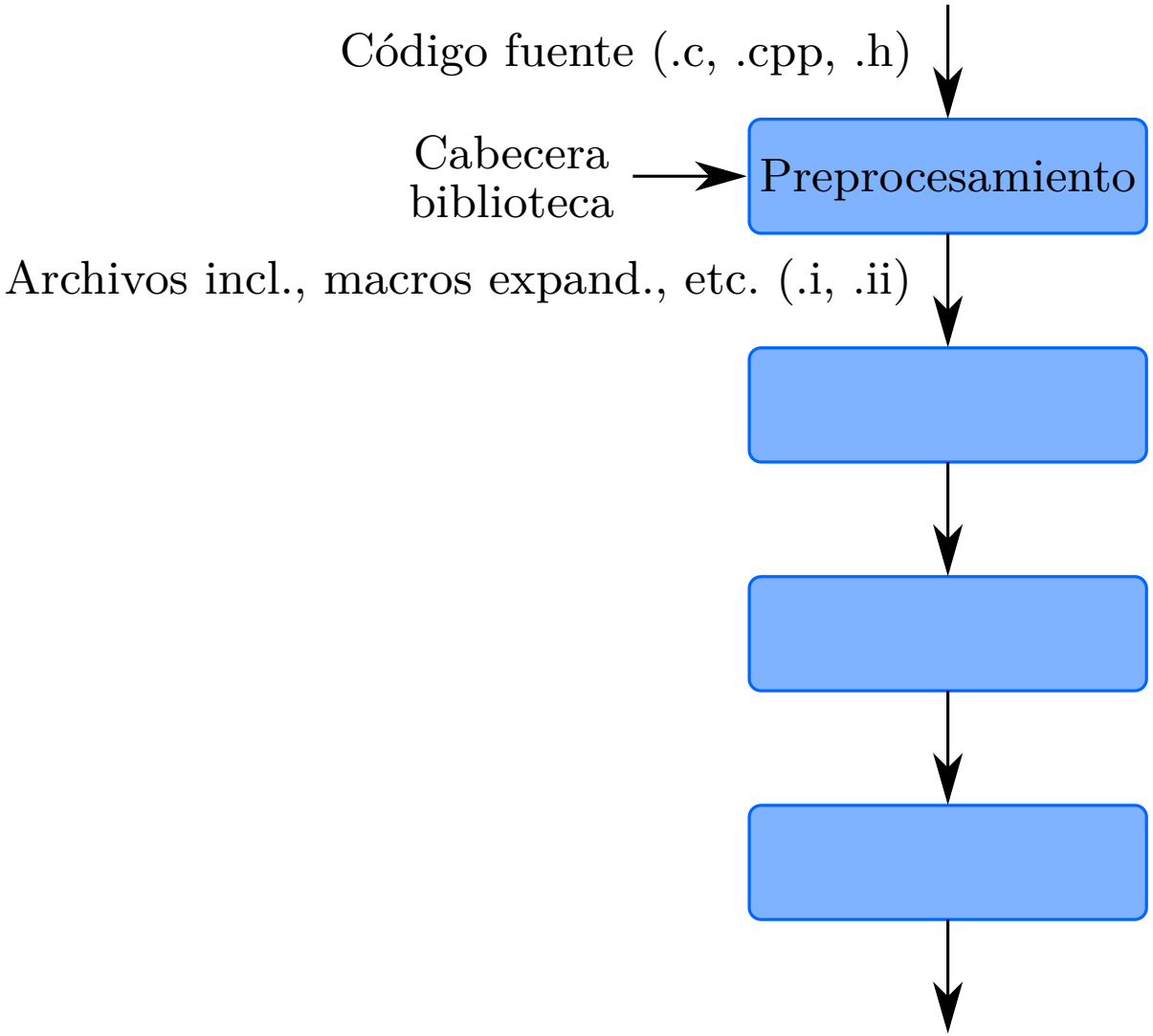
Etapas de compilación/construcción

El proceso de compilación/construcción involucra 4 etapas (preprocesamiento, compilación, ensamblado, y enlazado).



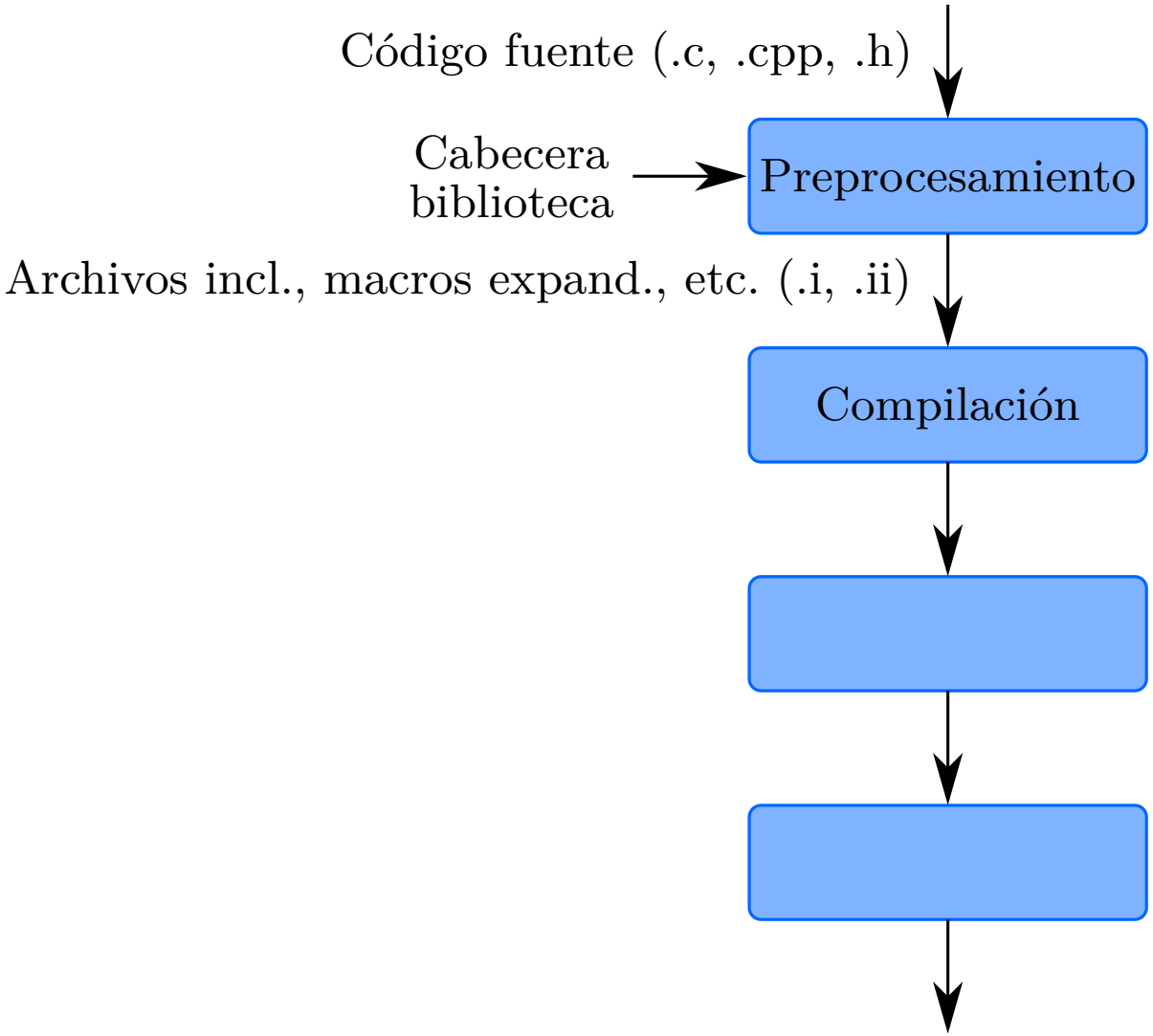
Etapas de compilación/construcción

El proceso de compilación/construcción involucra 4 etapas (preprocesamiento, compilación, ensamblado, y enlazado).



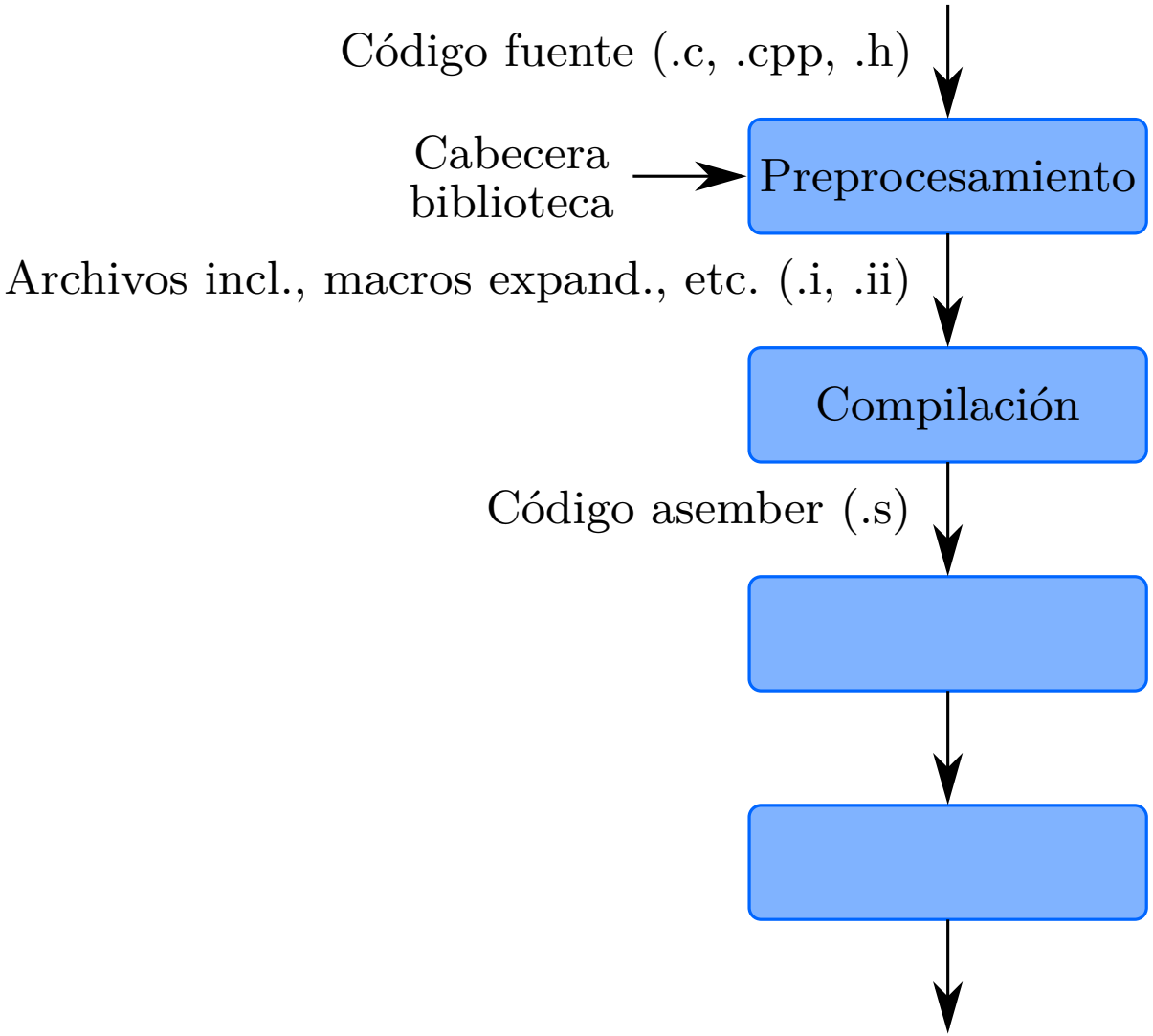
Etapas de compilación/construcción

El proceso de compilación/construcción involucra 4 etapas (preprocesamiento, compilación, ensamblado, y enlazado).



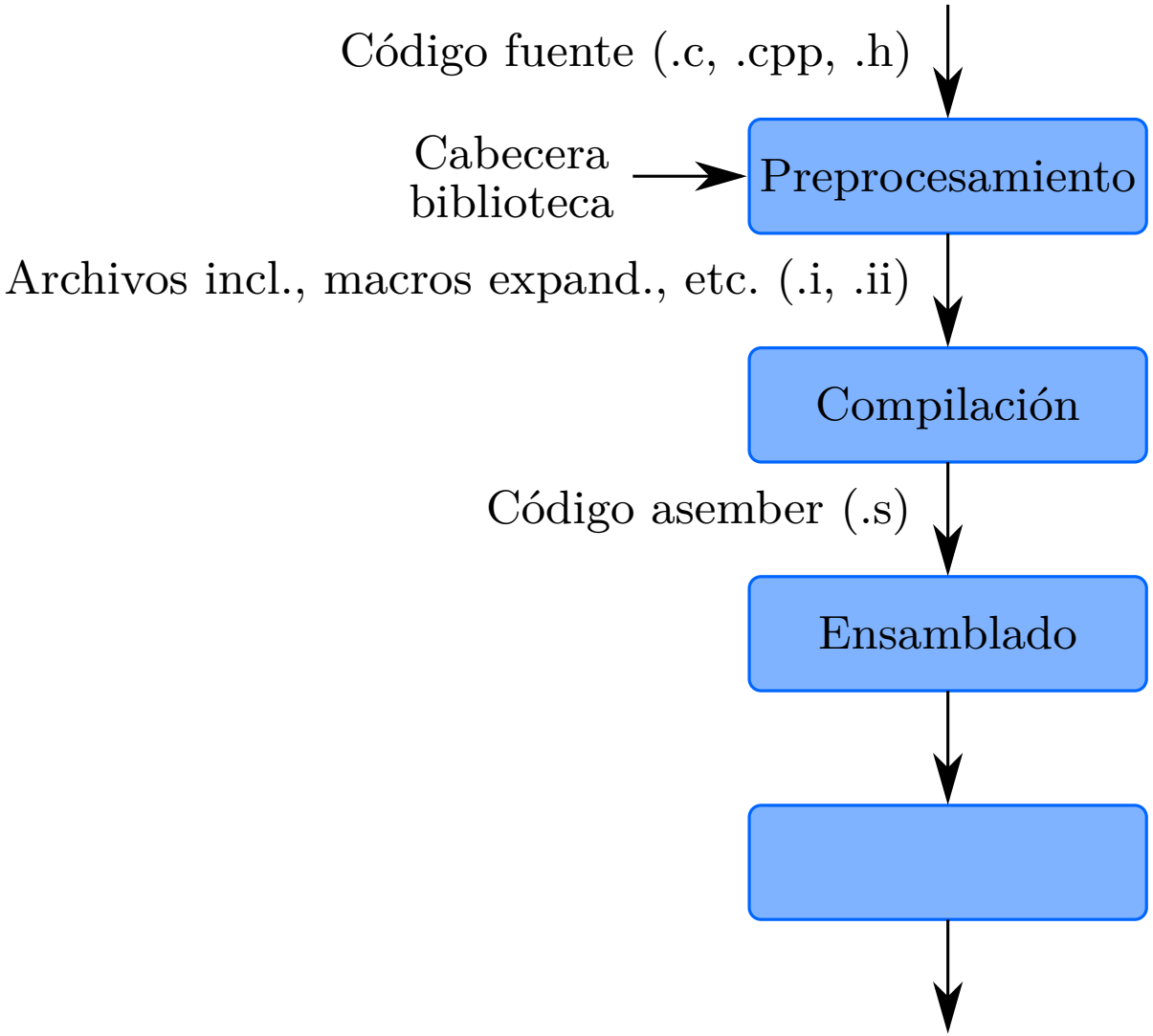
Etapas de compilación/construcción

El proceso de compilación/construcción involucra 4 etapas (preprocesamiento, compilación, ensamblado, y enlazado).



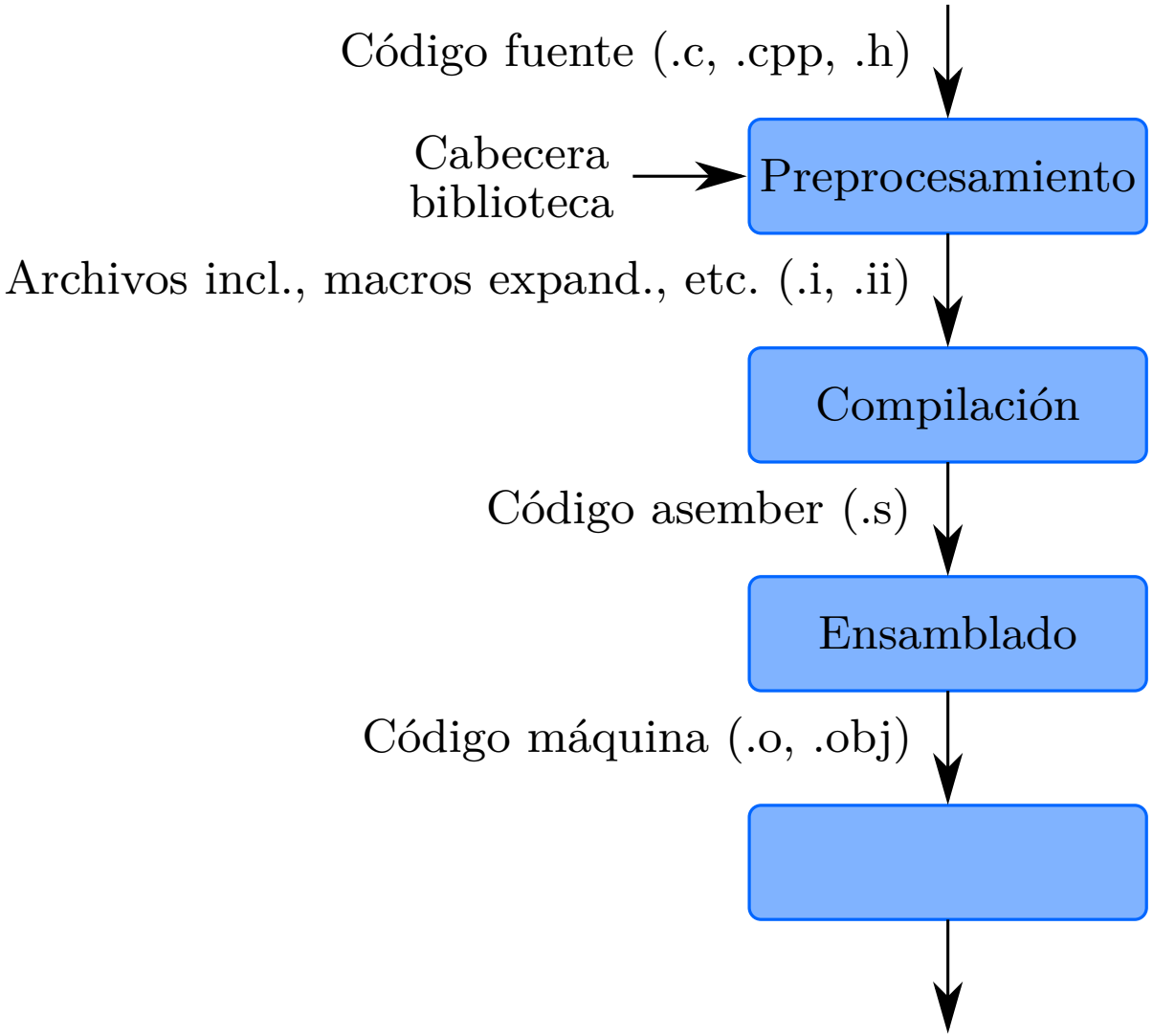
Etapas de compilación/construcción

El proceso de compilación/construcción involucra 4 etapas (preprocesamiento, compilación, ensamblado, y enlazado).



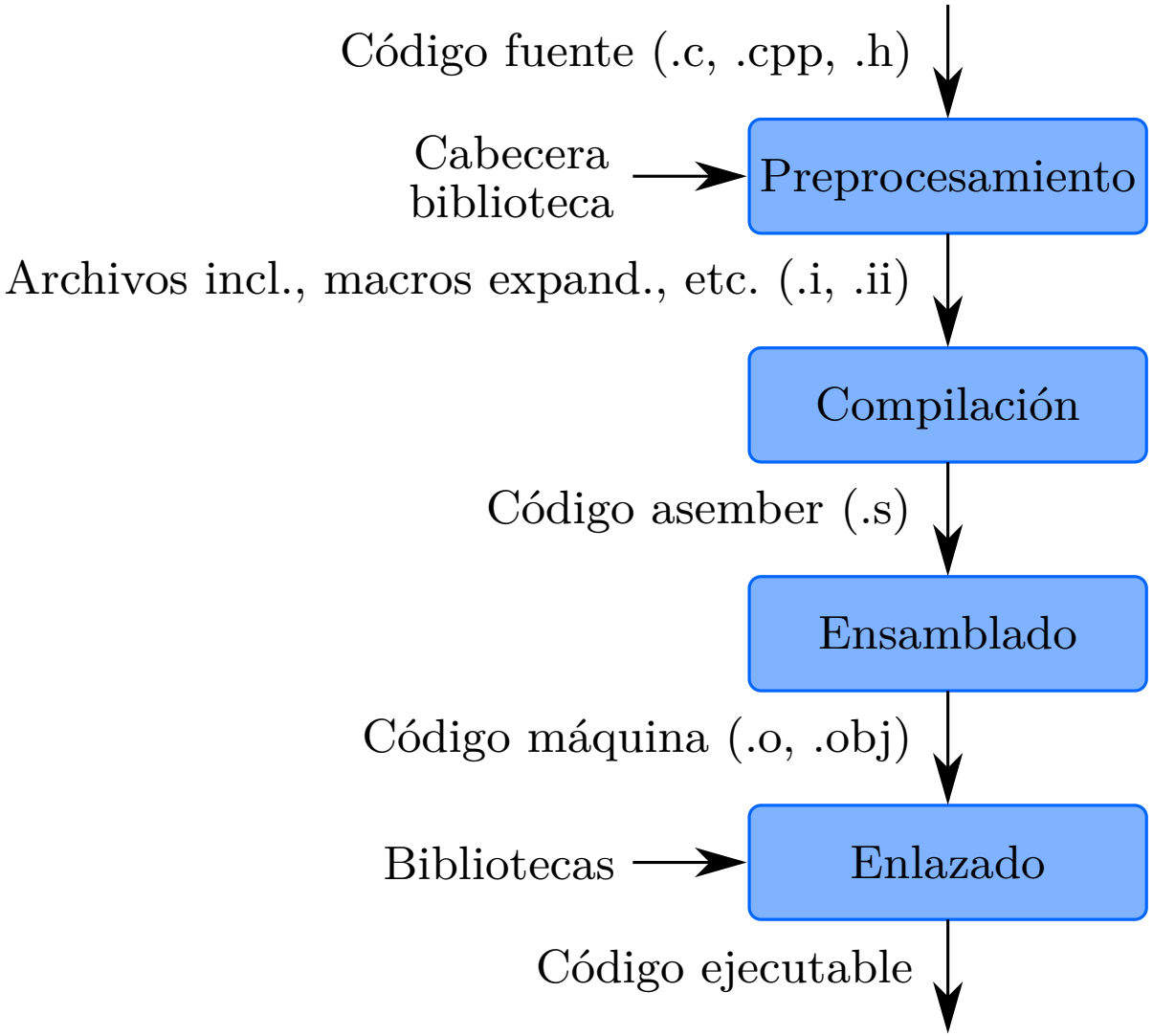
Etapas de compilación/construcción

El proceso de compilación/construcción involucra 4 etapas (preprocesamiento, compilación, ensamblado, y enlazado).



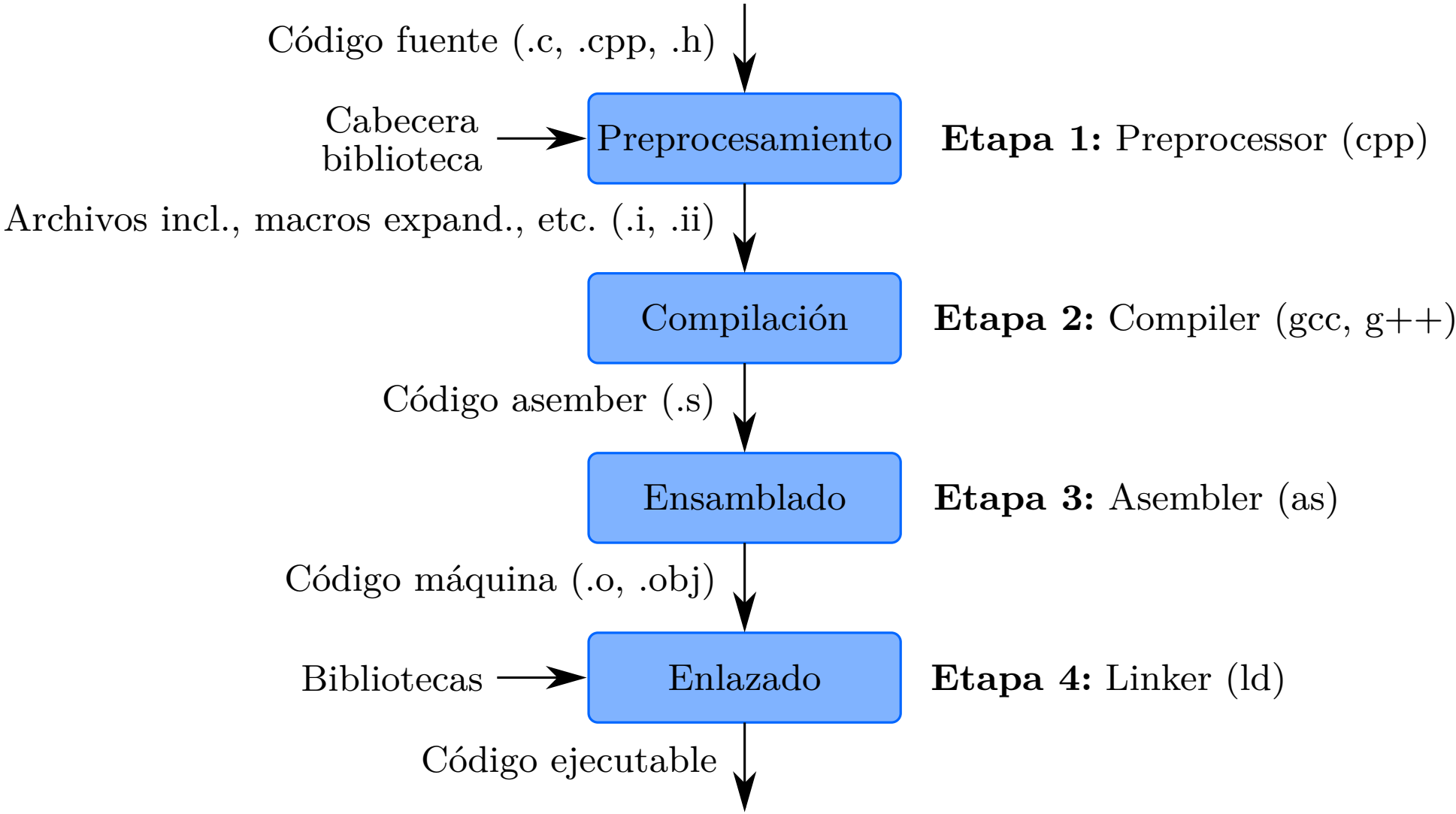
Etapas de compilación/construcción

El proceso de compilación/construcción involucra 4 etapas (preprocesamiento, compilación, ensamblado, y enlazado).



Etapas de compilación/construcción

El proceso de compilación/construcción involucra 4 etapas (preprocesamiento, compilación, ensamblado, y enlazado). Conjunto de herramientas: *toolchain*.



Preprocesado

test.c

```
1 #define TEST "Hola mundo!"
2 const char str[] = TEST;
```

Preprocesado

test.c

```
1 #define TEST "Hola mundo!"
2 const char str[] = TEST;
```

```
> gcc -E test.c
```

(O bien: `cpp test.c`)

Preprocesado

test.c

```
1 #define TEST "Hola mundo!"
2 const char str[] = TEST;
```

```
> gcc -E test.c
```

(O bien: `cpp test.c`)

Salida:

```
# 1 "test.c"
# 1 "<built-in>"
# 1 "<command-line>"
# 1 "/usr/include/stdc-predef.h" 1 3 4
# 1 "<command-line>" 2
# 1 "test.c"

const char str[] = "Hola, Mundo!";
```

Preprocesado

test.c

```
1 #define TEST "Hola mundo!"
2 const char str[] = TEST;
```

```
> gcc -E test.c
```

(O bien: `cpp test.c`)

Salida:

```
# 1 "test.c"
# 1 "<built-in>"
# 1 "<command-line>"
# 1 "/usr/include/stdc-predef.h" 1 3 4
# 1 "<command-line>" 2
# 1 "test.c"

const char str[] = "Hola, Mundo!";
```

(el preprocesador inserta líneas de registro de archivo fuente y el nro. de línea en la forma `#num-de-linea "archivo fuente"`)

Preprocesado

hola1.c

```c
#include <stdio.h>

/* Función main */
int main(void)
{
  /* Uso de cadena literal */
  printf("Hola mundo!\n");
  return 0;
}
```

hola2.c

```c
#include <stdio.h>
#define MENSAJE "Hola mundo!\n"

/* Función main */
int main(void)
{
  /* Uso de constante simbólica */
  printf(MENSAJE);
  return 0;
}
```

Preprocesado

hola1.c

```c
#include <stdio.h>

/* Función main */
int main(void)
{
  /* Uso de cadena literal */
  printf("Hola mundo!\n");
  return 0;
}
```

Preprocesado:

```
> gcc -E hola1.c > hola1.i
```

hola2.c

```c
#include <stdio.h>
#define MENSAJE "Hola mundo!\n"

/* Función main */
int main(void)
{
  /* Uso de constante simbólica */
  printf(MENSAJE);
  return 0;
}
```

Preprocesado:

```
> gcc -E hola2.c > hola2.i
```

Preprocesado

hola1.c

```c
#include <stdio.h>

/* Función main */
int main(void)
{
  /* Uso de cadena literal */
  printf("Hola mundo!\n");
  return 0;
}
```

Preprocesado:

```
> gcc -E hola1.c > hola1.i
```

hola2.c

```c
#include <stdio.h>
#define MENSAJE "Hola mundo!\n"

/* Función main */
int main(void)
{
  /* Uso de constante simbólica */
  printf(MENSAJE);
  return 0;
}
```

Preprocesado:

```
> gcc -E hola2.c > hola2.i
```

Comparar los archivos de salida.

Preprocesado

hola1.c

```c
#include <stdio.h>

/* Función main */
int main(void)
{
  /* Uso de cadena literal */
  printf("Hola mundo!\n");
  return 0;
}
```

hola2.c

```c
#include <stdio.h>
#define MENSAJE "Hola mundo!\n"

/* Función main */
int main(void)
{
  /* Uso de constante simbólica */
  printf(MENSAJE);
  return 0;
}
```

Preprocesado:

```
> gcc -E hola1.c > hola1.i
```

Preprocesado:

```
> gcc -E hola2.c > hola2.i
```

Comparar los archivos de salida.

Observar: constantes simbólicas, comentarios, archivos cabecera (includes).

Compilado, ensamblado y enlazado

Compilar el archivo de salida del preprocesador:

```
> gcc -Wall -S hola.i
```

[entrada: `hola.i` – salida: `hola.s`]

Compilado, ensamblado y enlazado

Compilar el archivo de salida del preprocesador:

```
> gcc -Wall -S hola.i
```

[entrada: `hola.i` – salida: `hola.s`]

Ensamblado:

```
> gcc -c hola.s -o hola.o
```

[entrada: `hola.s` – salida: `hola.o`]

(O bien: `as hola.s -o hola.o`)

Compilado, ensamblado y enlazado

Compilar el archivo de salida del preprocesador:

```
> gcc -Wall -S hola.i
```

[entrada: `hola.i` – salida: `hola.s`]

Ensamblado:

```
> gcc -c hola.s -o hola.o
```

[entrada: `hola.s` – salida: `hola.o`]

(O bien: `as hola.s -o hola.o`)

Enlazado:

```
>gcc hola.o -o hola
```

(Se puede utilizar también `ld`)

Construcción paso a paso – Resumen

Ejecutar diferentes etapas de construcción:

- `gcc -E`: Preprocesamiento sin compilación
- `gcc -S`: Compilación sin ensamblado
- `gcc -c`: Preprocesamiento, compilación y ensablado sin enlazado

Construcción paso a paso – Resumen

Ejecutar diferentes etapas de construcción:

- `gcc -E`: Preprocesamiento sin compilación
- `gcc -S`: Compilación sin ensamblado
- `gcc -c`: Preprocesamiento, compilación y ensablado sin enlazado

Ver página de manual de `gcc` (`> man gcc`)

Construcción paso a paso – Resumen

Ejecutar diferentes etapas de construcción:

- `gcc -E`: Preprocesamiento sin compilación
- `gcc -S`: Compilación sin ensamblado
- `gcc -c`: Preprocesamiento, compilación y ensablado sin enlazado

Ver página de manual de `gcc` (`> man gcc`)

Flag `--save-temps` genera archivos intermedios

```
> gcc --save-temps hola.c -o hola
```

(Ver flag `-v`)

Construcción paso a paso – Resumen

Ejecutar diferentes etapas de construcción:

- `gcc -E`: Preprocesamiento sin compilación
- `gcc -S`: Compilación sin ensamblado
- `gcc -c`: Preprocesamiento, compilación y ensablado sin enlazado

Ver página de manual de `gcc` (`> man gcc`)

Flag `--save-temps` genera archivos intermedios

```
> gcc --save-temps hola.c -o hola
```

(Ver flag `-v`)

Otros flags: `-std=c90`, `-Wall`, `-Werror`

Construcción paso a paso – Archivos de salida

```
> file hola.s
hola.s: assembler source, ASCII text
```

[`file hola.c`, `file hola.i` (?)]

Construcción paso a paso – Archivos de salida

```
> file hola.s
hola.s: assembler source, ASCII text
```

[`file hola.c`, `file hola.i` (?)]

```
> file hola.o
hola.o: ELF 64-bit LSB relocatable, x86-64,
version 1 (SYSV), not stripped
```

Construcción paso a paso – Archivos de salida

```
> file hola.s
hola.s: assembler source, ASCII text
```

[`file hola.c`, `file hola.i` (?)]

```
> file hola.o
hola.o: ELF 64-bit LSB relocatable, x86-64,
version 1 (SYSV), not stripped
```

```
> file hola
hola ELF 64-bit LSB executable, x86-64, version 1
(SYSV), dynamically linked, interpreter /lib64/ld-
linux-x86-64.so.2, for GNU/Linux 2.6.32, BuildID[sha1]
=620d3c9fadabd53755c0a647c0a43a172481a6f8, not stripped
```

Construcción paso a paso – Archivos de salida

```
> file hola.s
hola.s: assembler source, ASCII text
```

[`file hola.c`, `file hola.i` (?)]

```
> file hola.o
hola.o: ELF 64-bit LSB relocatable, x86-64,
version 1 (SYSV), not stripped
```

```
> file hola
hola ELF 64-bit LSB executable, x86-64, version 1
(SYSV), dynamically linked, interpreter /lib64/ld-
linux-x86-64.so.2, for GNU/Linux 2.6.32, BuildID[sha1]
=620d3c9fadabd53755c0a647c0a43a172481a6f8, not stripped
```

```
> ldd hola
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6
```

Construcción paso a paso – Archivos de salida

```
> file hola.s
hola.s: assembler source, ASCII text
```

[`file hola.c`, `file hola.i` (?)]

```
> file hola.o
hola.o: ELF 64-bit LSB relocatable, x86-64,
version 1 (SYSV), not stripped
```

```
> file hola
hola ELF 64-bit LSB executable, x86-64, version 1
(SYSV), dynamically linked, interpreter /lib64/ld-
linux-x86-64.so.2, for GNU/Linux 2.6.32, BuildID[sha1]
=620d3c9fadabd53755c0a647c0a43a172481a6f8, not stripped
```

```
> ldd hola
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6
```

ELF: Executable and Linkable Format

Construcción paso a paso – Archivos de salida

```
> file hola.s
hola.s: assembler source, ASCII text
```

[`file hola.c`, `file hola.i` (?)]

```
> file hola.o
hola.o: ELF 64-bit LSB relocatable, x86-64,
version 1 (SYSV), not stripped
```

```
> file hola
hola ELF 64-bit LSB executable, x86-64, version 1
(SYSV), dynamically linked, interpreter /lib64/ld-
linux-x86-64.so.2, for GNU/Linux 2.6.32, BuildID[sha1]
=620d3c9fadabd53755c0a647c0a43a172481a6f8, not stripped
```

```
> ldd hola
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6
```

ELF: Executable and Linkable Format

Otros comandos a probar: `hexdump -C`, `objdump -x`, `readelf -d`

El preprocesador

El preprocesamiento ocurre antes de la compilación de un programa.

El preprocesador

El preprocesamiento ocurre antes de la compilación de un programa.

Algunas de la acciones son:

1. inclusión de otros archivos dentro del archivo a compilar
2. definición de constantes simbólicas y macros
3. compilación condicional del código del programa

El preprocesador – Inclusión

La directiva del preprocesador `#include` provoca la inclusión de una copia del archivo especificado en el lugar de la directiva.

El preprocesador – Inclusión

La directiva del preprocesador **#include** provoca la inclusión de una copia del archivo especificado en el lugar de la directiva.

Tiene dos formas:

```
1 #include <nombre de archivo>
2 #include "nombre de archivo"
```

El preprocesador – Inclusión

La directiva del preprocesador **`#include`** provoca la inclusión de una copia del archivo especificado en el lugar de la directiva.

Tiene dos formas:

```
1 #include <nombre de archivo>
2 #include "nombre de archivo"
```

Difieren en la ubicación en la que el preprocesador busca el archivo a incluir:

1. Nombre del archivo entre llaves angulares (< y >) se utiliza por los *encabezados de la biblioteca estándar*.
2. Nombre del archivo entre comillas: busca el archivo en el mismo directorio en donde se encuentra el archivo que va a compilarse.

El preprocesador – Inclusión

La directiva del preprocesador **#include** provoca la inclusión de una copia del archivo especificado en el lugar de la directiva.

Tiene dos formas:

```
1 #include <nombre de archivo>
2 #include "nombre de archivo"
```

Difieren en la ubicación en la que el preprocesador busca el archivo a incluir:

1. Nombre del archivo entre llaves angulares (< y >) se utiliza por los *encabezados de la biblioteca estándar*.
2. Nombre del archivo entre comillas: busca el archivo en el mismo directorio en donde se encuentra el archivo que va a compilarse.

Las declaraciones que se incluyen en archivos de cabecera son de estructuras y uniones, enumeraciones y prototipos de funciones.

El preprocesador – Contantes simbólicas

La directiva `#define` crea constantes simbólicas (constantes representadas por símbolos) y macros (operaciones definidas como símbolos). El formato es:

```
#define identificador texto de reemplazo
```

El preprocesador – Contantes simbólicas

La directiva `#define` crea constantes simbólicas (constantes representadas por símbolos) y macros (operaciones definidas como símbolos). El formato es:

```
#define identificador texto de reemplazo
```

Hace que las ocurrencias subsecuentes del identificador sean reempazadas con el texto de reemplazo. Por ejemplo:

```
#define PI 3.14159
```

reemplaza todas las ocurrencias de la constantes simbólica `PI` con la constante numérica `3.14159`.

El preprocesador – Macros

- Una macro es un identificador definido dentro de una directiva de preprocesador `#define`.
- Como en las constantes simbólicas, el identificador de la macro se reemplaza en el programa con el texto de reemplazo.

El preprocesador – Macros

- Una macro es un identificador definido dentro de una directiva de preprocesador `#define`.
- Como en las constantes simbólicas, el identificador de la macro se reemplaza en el programa con el texto de reemplazo.

Las macros se puede definir con o sin argumentos.

- Una macro sin argumentos se procesa como una constante simbólica.
- En una macro con argumentos, los argumentos se sustituyen dentro del texto de reemplazo, y después se desarrolla la macro.

El preprocesador – Macros

- Una macro es un identificador definido dentro de una directiva de preprocesador `#define`.
- Como en las constantes simbólicas, el identificador de la macro se reemplaza en el programa con el texto de reemplazo.

Las macros se puede definir con o sin argumentos.

- Una macro sin argumentos se procesa como una constante simbólica.
- En una macro con argumentos, los argumentos se sustituyen dentro del texto de reemplazo, y después se desarrolla la macro.

Por ejemplo:

```
#define AREA_CIRCULO( x ) ( (PI) * (x) * (x) )
```

El preprocesador – Macros

- Una macro es un identificador definido dentro de una directiva de preprocesador `#define`.
- Como en las constantes simbólicas, el identificador de la macro se reemplaza en el programa con el texto de reemplazo.

Las macros se puede definir con o sin argumentos.

- Una macro sin argumentos se procesa como una constante simbólica.
- En una macro con argumentos, los argumentos se sustituyen dentro del texto de reemplazo, y después se desarrolla la macro.

Por ejemplo:

```
#define AREA_CIRCULO( x ) ( (PI) * (x) * (x) )
```

Cuando apareza `AREA_CIRCULO(y)` en el archivo, el valor de `x` se sustituirá por `y` dentro del texto de reemplazo.

El preprocesador – Macros

Por ejemplo, la línea:

```
area = AREA_CIRCULO(4);
```

se desarrolla como:

```
area = ( ( 3.14159 ) * (4) * (4) );
```

y el valor de la expresión se evalúa y se asigna a la variable `area`.

El preprocesador – Macros

Por ejemplo, la línea:

```
area = AREA_CIRCULO(4);
```

se desarrolla como:

```
area = ( ( 3.14159 ) * (4) * (4) );
```

y el valor de la expresión se evalúa y se asigna a la variable `area`.

Los paréntesis alrededor de cada `x` dentro del texto de reemplazo fuerza el orden apropiado de evaluación, cuando el argumento de la macro es una expresión:

```
area = AREA_CIRCULO(c + 2);
```

se desarrolla como:

```
area = AREA_CIRCULO( (3.14159) * (c + 2) * (c + 2) );
```

El preprocesador – Macros

Por ejemplo, la línea:

```
area = AREA_CIRCULO(4);
```

se desarrolla como:

```
area = ( ( 3.14159 ) * (4) * (4) );
```

y el valor de la expresión se evalúa y se asigna a la variable `area`.

Los paréntesis alrededor de cada `x` dentro del texto de reemplazo fuerza el orden apropiado de evaluación, cuando el argumento de la macro es una expresión:

```
area = AREA_CIRCULO(c + 2);
```

se desarrolla como:

```
area = AREA_CIRCULO( (3.14159) * (c + 2) * (c + 2) );
```

Las macros pueden tener más de un argumento. Por ejemplo:

```
#define AREA_RECTANGULO(x, y) ( (x) * (y) )
```

El preprocesador – Macros con valor

dtestval.c

```
1 #include <stdio.h>
2
3 int main(void)
4 {
5   printf("El valor de NUM es %d\n", NUM);
6   return 0;
7 }
```

El preprocesador – Macros con valor

dtestval.c

```c
#include <stdio.h>

int main(void)
{
  printf("El valor de NUM es %d\n", NUM);
  return 0;
}
```

```
> gcc -Wall -DNUM=100 dtestval.c
> ./a.out
El valor de NUM es 100
```

(-DNAME=VALUE)

El preprocesador – Macros con valor

dtestval.c

```c
#include <stdio.h>

int main(void)
{
  printf("El valor de NUM es %d\n", NUM);
  return 0;
}
```

```
> gcc -Wall -DNUM=100 dtestval.c
> ./a.out
El valor de NUM es 100
```

(-DNAME=VALUE)

```
> gcc -Wall -DNUM="2+2" dtestval.c
> ./a.out
El valor de NUM es 4
```

El preprocesador – Macros con valor

dtestval.c

```c
#include <stdio.h>

int main(void)
{
  printf("El valor de NUM es %d\n", NUM);
  return 0;
}
```

```
> gcc -Wall -DNUM=100 dtestval.c
> ./a.out
El valor de NUM es 100
```

(`-DNAME=VALUE`)

```
> gcc -Wall -DNUM="2+2" dtestval.c
> ./a.out
El valor de NUM es 4
```

(Macros predefinidas: `cpp -dM /dev/null`, p.e.: `GNUC`)

El preprocesador – Macros con valor

dtestval1.c

```
1 #include <stdio.h>
2
3 int main(void)
4 {
5   printf("Diez veces NUM es %d\n", 10 * (NUM));
6   return 0;
7 }
```

El preprocesador – Macros con valor

dtestval1.c

```c
#include <stdio.h>

int main(void)
{
  printf("Diez veces NUM es %d\n", 10 * (NUM));
  return 0;
}
```

```
> gcc -Wall -DNUM="2+2" dtestval1.c
> ./a.out
El valor de NUM es 40
```

El preprocesador – Macros con valor

dtestval1.c

```c
#include <stdio.h>

int main(void)
{
  printf("Diez veces NUM es %d\n", 10 * (NUM));
  return 0;
}
```

```
> gcc -Wall -DNUM="2+2" dtestval1.c
> ./a.out
El valor de NUM es 40
```

¿Qué valor se imprime si no se ponen los paréntesis en la macro?

El preprocesador – Macros con valor

dtestval1.c

```c
#include <stdio.h>

int main(void)
{
  printf("Diez veces NUM es %d\n", 10 * (NUM));
  return 0;
}
```

```
> gcc -Wall -DNUM="2+2" dtestval1.c
> ./a.out
El valor de NUM es 40
```

¿Qué valor se imprime si no se ponen los paréntesis en la macro?

Valor por defecto de una macro

```
> gcc -Wall -DNUM dtestval.c
> ./a.out
El valor de NUM es 1
```

El preprocesador – Macros con valor

dtestval1.c

```c
#include <stdio.h>

int main(void)
{
  printf("Diez veces NUM es %d\n", 10 * (NUM));
  return 0;
}
```

```
> gcc -Wall -DNUM="2+2" dtestval1.c
> ./a.out
El valor de NUM es 40
```

¿Qué valor se imprime si no se ponen los paréntesis en la macro?

Valor por defecto de una macro

```
> gcc -Wall -DNUM dtestval.c
> ./a.out
El valor de NUM es 1
```

Una macro vacía `-DNUM=""` queda definida (`#ifdef`) pero no se expande a nada.

El preprocesador – Compilación condicional

dtest.c

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5  #ifdef TEST
6    printf("Modo test\n");
7  #endif
8    printf("Ejecutando...\n");
9    return 0;
10 }
```

El preprocesador – Compilación condicional

dtest.c

```c
#include <stdio.h>

int main(void)
{
#ifdef TEST
  printf("Modo test\n");
#endif
  printf("Ejecutando...\n");
  return 0;
}
```

```
> gcc -Wall dtest.c
> ./a.out
Ejecutando...
```

El preprocesador – Compilación condicional

dtest.c

```c
#include <stdio.h>

int main(void)
{
#ifdef TEST
  printf("Modo test\n");
#endif
  printf("Ejecutando...\n");
  return 0;
}
```

```
> gcc -Wall dtest.c
> ./a.out
Ejecutando...
```

Macro definida desde la línea de comandos

```
> gcc -Wall -DTEST dtest.c
> ./a.out
Modo test
Ejecutando...
```

El preprocesador – Compilación condicional

dtest.c

```c
#include <stdio.h>

int main(void)
{
#ifdef TEST
  printf("Modo test\n");
#endif
  printf("Ejecutando...\n");
  return 0;
}
```

```
> gcc -Wall dtest.c
> ./a.out
Ejecutando...
```

Macro definida desde la línea de comandos

```
> gcc -Wall -DTEST dtest.c
> ./a.out
Modo test
Ejecutando...
```

(probar utilizando `gcc -E`)