

## Fases de la Traducción y Errores

### 7.1. Objetivos

Este trabajo tiene como objetivo identificar las **fases del proceso de traducción o Build** y los **posibles errores asociados a cada fase**.

Para lograr esa identificación se ejecutan las **fases de traducción una a una, se detectan y corrigen errores, y se registran las conclusiones en readme.md**.

No es un trabajo de desarrollo; es más, el programa que usamos como ejemplo es simple, similar a `hello.c` pero con errores que se deben corregir. La complejidad está en la **identificación y comprensión de las etapas y sus productos**.



### 7.2. Temas

- Fases de traducción.
- Preprocesamiento.
- Compilación.
- Ensamblado.
- Vinculación (Link).
- Errores en cada fase.

- Compilación separada.

## 7.3. Tareas

1. La primera tarea es investigar las funcionalidades y opciones que su implementación, es decir su compilador, presenta para limitar el inicio y fin de las fases de traducción.
2. La siguiente tarea es poner en práctica lo que se investigó. Para eso se debe transcribir al `readme.md` cada **comando ejecutado** y su **resultado o error** correspondiente a la siguiente secuencia de pasos. También en `readme.md` se vuelcan las conclusiones y se resuelven los puntos solicitados. Para claridad, mantener en `readme.md` la misma numeración de la secuencia de pasos.

### 7.3.1. Secuencia de Pasos

Se parte de un archivo fuente que es corregido y refinado en sucesivos pasos. Es importante no saltarse pasos para mantener la correlación, ya que el estado dejado por el paso anterior es necesario para el siguiente.

1. Preprocesador
  - a. Escribir `hello2.c`, que es una variante de `hello.c`:

```
#include <stdio.h>

int/*medio*/main(void){
    int i=42;
    printf("La respuesta es %d\n");
```

- b. Preprocesar `hello2.c`, no compilar, y generar `hello2.i`. Analizar su contenido. ¿Qué conclusiones saca?
- c. Escribir `hello3.c`, una nueva variante:

```
int printf(const char * restrict s, ...);

int main(void){
    int i=42;
    printf("La respuesta es %d\n");
```

- d. Investigar e indicar la semántica de la primera línea.

- e. Preprocesar `hello3.c`, no compilar, y generar `hello3.i`. Buscar diferencias entre `hello3.c` y `hello3.i`.

## 2. Compilación

- a. Compilar el resultado y generar `hello3.s`, no ensamblar.
- b. Corregir solo los errores, no los *warnings*, en el nuevo archivo `hello4.c` y empezar de nuevo, generar `hello4.s`, no ensamblar.
- c. Leer `hello4.s`, investigar sobre lenguaje ensamblador, e indicar de formar sintética cual es el objetivo de ese código.
- d. Ensamblar `hello4.s` en `hello4.o`, no vincular.

## 3. Vinculación

- a. Vincular `hello4.o` con la biblioteca estándar y generar el ejecutable.
- b. Corregir en `hello5.c` y generar el ejecutable. Solo corregir lo necesario para que vincule.
- c. Ejecutar y analizar el resultado.

## 4. Corrección de Bug

- a. Corregir en `hello6.c` y empezar de nuevo; verificar que funciona como se espera.

## 5. Remoción de prototipo

- a. Escribir `hello7.c`, una nueva variante:

```
int main(void){
    int i=42;
    printf("La respuesta es %d\n", i);
}
```

- b. Explicar porqué funciona; para eso, considerar las siguientes preguntas:
  - i. ¿Arroja *error* o *warning*?
  - ii. ¿Qué es un *prototipo* y de qué maneras se puede generar?
  - iii. ¿Qué es una *declaración implícita de una función*?
  - iv. ¿Qué indica la *especificación*?

- v. ¿Cómo se comportan las *principales implementaciones*?
- vi. ¿Qué es una función *built-in*?
- vii. ¿Conjeture la razón por la cual gcc se comporta como se comporta?  
¿Va realmente contra la especificación?

## 6. Compilación Separada: Contratos y Módulos

- a. Escribir `studio1.c` (sí, `studio1`, no `stdio`) y `hello8.c`.

La unidad de traducción `studio1.c` tiene una implementación de la función `prntf`, que es solo un *wrapper*<sup>1</sup> de la función estándar `printf`:

```
void prntf(const char* s, int i){
    printf("La respuesta es %d\n", i);
}
```

La unidad de traducción `hello8.c`, muy similar a `hello4.c`, invoca a `prntf`, pero no incluye ningún header.

```
int main(void){
    int i=42;
    prntf("La respuesta es %d\n", i);
}
```

- b. Investigar como en su entorno de desarrollo puede generar un programa ejecutable que se base en las dos unidades de traducción (i.e., archivos fuente, archivos con extensión `.c`).  
Luego generar ese ejecutable y probarlo.
- c. Responder ¿qué ocurre si eliminamos o agregamos argumentos a la invocación de `prntf`? Justifique.
- d. Revisitar el punto anterior, esta vez utilizando un contrato de interfaz en un archivo header.
- i. Escribir el contrato en `studio.h`.

```
#ifndef _STUDIO_H_INCULDED_
#define _STUDIO_H_INCULDED_
```

<sup>1</sup> [https://en.wikipedia.org/wiki/Wrapper\\_function](https://en.wikipedia.org/wiki/Wrapper_function)

```
void prntf(const char*, int);

#endif
```

- ii. Escribir `hello9.c`, un cliente que sí incluye el contrato.

```
#include "studio.h" // Interfaz que importa

int main(void){
    int i=42;
    prntf("La respuesta es %d\n", i);
}
```

- iii. Escribir `studio2.c`, el proveedor que sí incluye el contrato.

```
#include "studio.h" // Interfaz que exporta
#include <stdio.h> // Interfaz que importa

void prntf(const char* s, int i){
    printf("La respuesta es %d\n", i);
}
```

- iv. Responder: ¿Qué ventaja da incluir el contrato en los clientes y en el proveedor.



### Crédito extra

Investigue sobre *bibliotecas*. ¿Qué son? ¿Se pueden distribuir? ¿Son portables? ¿Cuáles son sus ventajas y desventajas?.

Desarrolle y utilice la biblioteca `studio`.

## 7.4. Restricciones

- El programa ejemplo debe enviar por `stdout` la frase `La respuesta es 42`, el valor 42 debe surgir de una variable.

## 7.5. Productos

```
DD-FasesErrores
|-- readme.md
```

```
|-- hello2.c  
|-- hello3.c  
|-- hello4.c  
|-- hello5.c  
|-- hello6.c  
|-- hello7.c  
|-- hello8.c  
|-- hello9.c  
|-- studio1.c  
|-- studio.h  
`-- studio2.c
```

---

# 8

## Módulo Stack

---

### 8.1. Objetivos

Construir dos implementaciones del Módulo Stack de `int`s`.

### 8.2. Temas

- Módulos.
- Interfaz.
- Stack.
- Unit tests.
- `assert`
- Reserva estática de memoria.
- Ocultamiento de información.
- Encapsulamiento.
- Precondiciones.
- Poscondiciones.
- Call stack.
- heap.
- Reserva dinámica de memoria.
- Punteros.
- `malloc`.
- `free`.

### 8.3. Tareas

1. Analizar el stack de la sección 4.3 de [\[KR1988\]](#).
2. Codificar la interfaz `stackModule.h` para que incluya las operaciones:
  - a. Push.
  - b. Pop.
  - c. IsEmpty.
  - d. IsFull.
3. Escribir en la interfaz `stackModule.h` comentarios que incluya *especificaciones* y *pre* y *poscondiciones* de las operaciones.
4. Codificar los unit tests en `stackModuleTest.c`.
5. Codificar una implementación contigua y estática en `stackModuleContiguousStatic.c`.
6. Probar `stackModuleContiguousStatic.c` con `stackModuleTest.c`.
7. Codificar una implementación enlazada y dinámica en `stackModuleLinkedDynamic.c`.
8. Probar `stackModuleLinkedDynamic.c` con `stackModuleTest.c`.
9. Probar `StackDynamic.c` con `StackTest`.
10. Construir una tabla comparativa a modo de *benchmark* que muestre el tiempo de procesamiento para cada una de las dos implementaciones.
11. Diseñar el archivo `makefile` para que construya una, otra o ambas implementaciones, y para que ejecute las pruebas.
12. Responder:
  - a. ¿Cuál es la mejor implementación? Justifique.
  - b. ¿Qué cambios haría para que no haya precondiciones? ¿Qué implicancia tiene el cambio?
  - c. ¿Qué cambios haría en el diseño para que el stack sea genérico, es decir permita elementos de otros tipos que no sean `int`? ¿Qué implicancia tiene el cambio?
  - d. Proponga un nuevo diseño para que el módulo pase a ser un *tipo de dato*, es decir, permita a un programa utilizar más de un stack.



## 8.4. Restricciones

- En `stackModule.h`:
  - Aplicar guardas de inclusión.
  - Declarar `typedef int StackItem;`
- En `stackModuleTest.c` incluir `assert.h` y aplicar `assert`.
- En ambas implementaciones utilizar `static` para aplicar encapsulamiento.
- En la implementación contigua y estática:
  - No utilizar índices, sí aritmética punteros.
  - Aplicar el *idiom* para stacks.
- En la implementación enlazada y dinámica:
  - Invocar a `malloc` y a `free`.
  - No utilizar el operador `sizeof( tipo )`, sí `sizeof expresión`.

## 8.5. Productos

- Sufijo del nombre de la carpeta: `stackModule`.
- `/Readme.md`
  - Benchmark.
  - Preguntas y Respuestas.
- `/StackModule.h`.
- `/StackModuleTest.c`
- `/StackModuleContiguousStatic.c`
- `/StackModuleLinkedDynamic.c`
- `/Makefile`



---

# **Parte IV. Strings en Leguajes Formales y en Lenguajes de Progamación**

---