

# Informática II

## Repaso del lenguaje C

Gonzalo F. Pérez Paina



Universidad Tecnológica Nacional  
Facultad Regional Córdoba  
UTN-FRC

– 2018 –

# Introducción

## Un poco de historia

- ▶ Ideas provenientes de los lenguajes BCPL<sup>1</sup> (*Martin Richards, 1967*) y del lenguaje B (*Ken Thompson, 1970*) [lenguajes “sin tipo”].

---

<sup>1</sup>Basic Combined Programming Language

# Introducción

## Un poco de historia

- ▶ Ideas provenientes de los lenguajes BCPL<sup>1</sup> (*Martin Richards, 1967*) y del lenguaje B (*Ken Thompson, 1970*) [lenguajes “sin tipo”].
- ▶ C fue diseñado originalmente en 1972 para el sistema operativo UNIX en el DEC PDP-11 por *Dennis Ritchie* en los laboratorios Bell.

---

<sup>1</sup>Basic Combined Programming Language

# Introducción

## Un poco de historia

- ▶ Ideas provenientes de los lenguajes BCPL<sup>1</sup> (*Martin Richards, 1967*) y del lenguaje B (*Ken Thompson, 1970*) [lenguajes “sin tipo”].
- ▶ C fue diseñado originalmente en 1972 para el sistema operativo UNIX en el DEC PDP-11 por *Dennis Ritchie* en los laboratorios Bell.
- ▶ El primer libro de referencia fue *C Programming Language* (1978) de Brian Kernighan y Dennis Ritchie.

---

<sup>1</sup>Basic Combined Programming Language

# Introducción

## Un poco de historia

- ▶ Ideas provenientes de los lenguajes BCPL<sup>1</sup> (*Martin Richards, 1967*) y del lenguaje B (*Ken Thompson, 1970*) [lenguajes “sin tipo”].
- ▶ C fue diseñado originalmente en 1972 para el sistema operativo UNIX en el DEC PDP-11 por *Dennis Ritchie* en los laboratorios Bell.
- ▶ El primer libro de referencia fue *C Programming Language* (1978) de Brian Kernighan y Dennis Ritchie.
- ▶ En 1989 aparece el estándar ANSI C (ANSI<sup>2</sup>).

---

<sup>1</sup>Basic Combined Programming Language

<sup>2</sup>American National Standards Institute

# Introducción

## Un poco de historia

- ▶ Ideas provenientes de los lenguajes BCPL<sup>1</sup> (*Martin Richards, 1967*) y del lenguaje B (*Ken Thompson, 1970*) [lenguajes “sin tipo”].
- ▶ C fue diseñado originalmente en 1972 para el sistema operativo UNIX en el DEC PDP-11 por *Dennis Ritchie* en los laboratorios Bell.
- ▶ El primer libro de referencia fue *C Programming Language* (1978) de Brian Kernighan y Dennis Ritchie.
- ▶ En 1989 aparece el estándar ANSI C (ANSI<sup>2</sup>).
- ▶ En 1990 aparece el estándar ISO C (ISO<sup>3</sup>).

---

<sup>1</sup>Basic Combined Programming Language

<sup>2</sup>American National Standards Institute

<sup>3</sup>International Standards Organization

# Introducción

## Un poco de historia

- ▶ Ideas provenientes de los lenguajes BCPL<sup>1</sup> (*Martin Richards, 1967*) y del lenguaje B (*Ken Thompson, 1970*) [lenguajes “sin tipo”].
- ▶ C fue diseñado originalmente en 1972 para el sistema operativo UNIX en el DEC PDP-11 por *Dennis Ritchie* en los laboratorios Bell.
- ▶ El primer libro de referencia fue *C Programming Language* (1978) de Brian Kernighan y Dennis Ritchie.
- ▶ En 1989 aparece el estándar ANSI C (ANSI<sup>2</sup>).
- ▶ En 1990 aparece el estándar ISO C (ISO<sup>3</sup>).
- ▶ En 1999 aparece el estándar C99.

---

<sup>1</sup>Basic Combined Programming Language

<sup>2</sup>American National Standards Institute

<sup>3</sup>International Standards Organization

# Introducción

## Un poco de historia

- ▶ Ideas provenientes de los lenguajes BCPL<sup>1</sup> (*Martin Richards, 1967*) y del lenguaje B (*Ken Thompson, 1970*) [lenguajes “sin tipo”].
- ▶ C fue diseñado originalmente en 1972 para el sistema operativo UNIX en el DEC PDP-11 por *Dennis Ritchie* en los laboratorios Bell.
- ▶ El primer libro de referencia fue *C Programming Language* (1978) de Brian Kernighan y Dennis Ritchie.
- ▶ En 1989 aparece el estándar ANSI C (ANSI<sup>2</sup>).
- ▶ En 1990 aparece el estándar ISO C (ISO<sup>3</sup>).
- ▶ En 1999 aparece el estándar C99.
- ▶ El último estándar publicado de C es el C11 (ISO/IEC 9899:2011) (IEC<sup>4</sup>).

---

<sup>1</sup>Basic Combined Programming Language

<sup>2</sup>American National Standards Institute

<sup>3</sup>International Standards Organization

<sup>4</sup>International Electrotechnical Commission



# Introducción

## Algunas características

- ▶ Lenguaje de propósitos generales ampliamente utilizado.

# Introducción

## Algunas características

- ▶ Lenguaje de propósitos generales ampliamente utilizado.
- ▶ Tiene características de lenguajes de **bajo nivel**

# Introducción

## Algunas características

- ▶ Lenguaje de propósitos generales ampliamente utilizado.
- ▶ Tiene características de lenguajes de **bajo nivel**
- ▶ Lenguaje relativamente pequeño: ofrece sentencias de control sencillas y funciones.

# Introducción

## Algunas características

- ▶ Lenguaje de propósitos generales ampliamente utilizado.
- ▶ Tiene características de lenguajes de **bajo nivel**
- ▶ Lenguaje relativamente pequeño: ofrece sentencias de control sencillas y funciones.
- ▶ Permite **programación estructurada** y diseño modular.

# Introducción

## Algunas características

- ▶ Lenguaje de propósitos generales ampliamente utilizado.
- ▶ Tiene características de lenguajes de **bajo nivel**
- ▶ Lenguaje relativamente pequeño: ofrece sentencias de control sencillas y funciones.
- ▶ Permite **programación estructurada** y diseño modular.
- ▶ Si los programas siguen el estándar ISO el código es portátil entre plataformas y/o arquitecturas.

# Introducción

## Algunos inconvenientes

- ▶ No es un lenguaje **fuertemente tipado**. (ventaja o desventaja?)

# Introducción

## Algunos inconvenientes

- ▶ No es un lenguaje **fuertemente tipado**. (ventaja o desventaja?)
- ▶ Bastante permisivo con la conversión de datos.

# Introducción

## Algunos inconvenientes

- ▶ No es un lenguaje **fuertemente tipado**. (ventaja o desventaja?)
- ▶ Bastante permisivo con la conversión de datos.
- ▶ Su versatilidad permite crear programas difíciles de leer (código ofuscado)<sup>5</sup>.

---

<sup>5</sup>IOCCC: The International Obfuscated C Code Contest





# Introducción al lenguaje C

## Programas en lenguaje C

Los programas C consisten de módulos o piezas que se denominan funciones. Esto facilita evitar volver a inventar la rueda: *reutilización de software*.

# Introducción al lenguaje C

## Programas en lenguaje C

Los programas C consisten de módulos o piezas que se denominan funciones. Esto facilita evitar volver a inventar la rueda: *reutilización de software*.

## Aprender a programar en “C”

Consta de dos partes:

- ▶ Lenguaje C en sí mismo.
- ▶ Funciones de la biblioteca estándar C.

# Introducción al lenguaje C

## Programas en lenguaje C

Los programas C consisten de módulos o piezas que se denominan funciones. Esto facilita evitar volver a inventar la rueda: *reutilización de software*.

## Aprender a programar en “C”

Consta de dos partes:

- ▶ Lenguaje C en sí mismo.
- ▶ Funciones de la biblioteca estándar C.

Todos los sistemas C consisten, en general, en tres partes:

- ▶ el entorno,
- ▶ el lenguaje, y
- ▶ la biblioteca estándar C.

# Introducción al lenguaje C

## Programas en lenguaje C

Los programas C consisten de módulos o piezas que se denominan funciones. Esto facilita evitar volver a inventar la rueda: *reutilización de software*.

## Aprender a programar en “C”

Consta de dos partes:

- ▶ Lenguaje C en sí mismo.
- ▶ Funciones de la biblioteca estándar C.

Todos los sistemas C consisten, en general, en tres partes:

- ▶ el entorno,
- ▶ el lenguaje, y
- ▶ la biblioteca estándar C.

Los programas C casi siempre pasan a través de **seis fases** para su ejecución: *editar*, *preprocesar*, *compilar*, *enlazar*, *cargar*, *ejecutar*.

# Introducción al lenguaje C

## Programas de ejemplo

---

```
1 /* Primer programa en C */
2 #include <stdio.h>
3
4 int main(void)
5 {
6     printf("Hola mundo.\n");
7     return 0;
8 }
```

---

# Introducción al lenguaje C

## Programas de ejemplo

---

```
1 /* Primer programa en C */
2 #include <stdio.h>
3
4 int main(void)
5 {
6     printf("Hola mundo.\n");
7     return 0;
8 }
```

---

```
Hola mundo.
```

# Introducción al lenguaje C

## Programas de ejemplo

- ▶ Comentarios (¿para qué?)

---

```
1 /* Primer programa en C */
2 #include <stdio.h>
3
4 int main(void)
5 {
6     printf("Hola mundo.\n");
7     return 0;
8 }
```

---

```
Hola mundo.
```



# Introducción al lenguaje C

## Programas de ejemplo

---

```
1 /* Primer programa en C */
2 #include <stdio.h>
3
4 int main(void)
5 {
6     printf("Hola mundo.\n");
7     return 0;
8 }
```

---

```
Hola mundo.
```

- ▶ Comentarios (¿para qué?)
- ▶ Directriz del preprocesador (#)

# Introducción al lenguaje C

## Programas de ejemplo

---

```
1 /* Primer programa en C */
2 #include <stdio.h>
3
4 int main(void)
5 {
6     printf("Hola mundo.\n");
7     return 0;
8 }
```

---

```
Hola mundo.
```

- ▶ Comentarios (¿para qué?)
- ▶ Directriz del preprocesador (#)
- ▶ Archivos de cabecera/header (.h)

# Introducción al lenguaje C

## Programas de ejemplo

---

```
1 /* Primer programa en C */
2 #include <stdio.h>
3
4 int main(void)
5 {
6     printf("Hola mundo.\n");
7     return 0;
8 }
```

---

```
Hola mundo.
```

- ▶ Comentarios (¿para qué?)
- ▶ Directriz del preprocesador (#)
- ▶ Archivos de cabecera/header (.h)
- ▶ Biblioteca estándar (stdin, stdout)

# Introducción al lenguaje C

## Programas de ejemplo

---

```
1 /* Primer programa en C */
2 #include <stdio.h>
3
4 int main(void)
5 {
6     printf("Hola mundo.\n");
7     return 0;
8 }
```

---

```
Hola mundo.
```

- ▶ Comentarios (¿para qué?)
- ▶ Directriz del preprocesador (#)
- ▶ Archivos de cabecera/header (.h)
- ▶ Biblioteca estándar (stdin, stdout)
- ▶ Función main (paréntesis)

# Introducción al lenguaje C

## Programas de ejemplo

---

```
1 /* Primer programa en C */
2 #include <stdio.h>
3
4 int main(void)
5 {
6     printf("Hola mundo.\n");
7     return 0;
8 }
```

---

```
Hola mundo.
```

- ▶ Comentarios (¿para qué?)
- ▶ Directriz del preprocesador (#)
- ▶ Archivos de cabecera/header (.h)
- ▶ Biblioteca estándar (stdin, stdout)
- ▶ Función main (paréntesis)
- ▶ Parámetros y valor de retorno

# Introducción al lenguaje C

## Programas de ejemplo

---

```
1 /* Primer programa en C */
2 #include <stdio.h>
3
4 int main(void)
5 {
6     printf("Hola mundo.\n");
7     return 0;
8 }
```

---

```
Hola mundo.
```

- ▶ Comentarios (¿para qué?)
- ▶ Directriz del preprocesador (#)
- ▶ Archivos de cabecera/header (.h)
- ▶ Biblioteca estándar (stdin, stdout)
- ▶ Función main (paréntesis)
- ▶ Parámetros y valor de retorno
- ▶ Bloque (llaves)- Cuerpo de la función

# Introducción al lenguaje C

## Programas de ejemplo

---

```
1 /* Primer programa en C */
2 #include <stdio.h>
3
4 int main(void)
5 {
6     printf("Hola mundo.\n");
7     return 0;
8 }
```

---

```
Hola mundo.
```

- ▶ Comentarios (¿para qué?)
- ▶ Directriz del preprocesador (#)
- ▶ Archivos de cabecera/header (.h)
- ▶ Biblioteca estándar (stdin, stdout)
- ▶ Función main (paréntesis)
- ▶ Parámetros y valor de retorno
- ▶ Bloque (llaves)- Cuerpo de la función
- ▶ Parámetros y valor de devolución

# Introducción al lenguaje C

## Programas de ejemplo

---

```
1 /* Primer programa en C */
2 #include <stdio.h>
3
4 int main(void)
5 {
6     printf("Hola mundo.\n");
7     return 0;
8 }
```

---

```
Hola mundo.
```

- ▶ Comentarios (¿para qué?)
- ▶ Directriz del preprocesador (#)
- ▶ Archivos de cabecera/header (.h)
- ▶ Biblioteca estándar (stdin, stdout)
- ▶ Función main (paréntesis)
- ▶ Parámetros y valor de retorno
- ▶ Bloque (llaves)- Cuerpo de la función
- ▶ Parámetros y valor de devolución
- ▶ Enunciados (finaliza con ';'')



# Introducción al lenguaje C

## Programas de ejemplo

---

```
1 /* Primer programa en C */
2 #include <stdio.h>
3
4 int main(void)
5 {
6     printf("Hola mundo.\n");
7     return 0;
8 }
```

---

```
Hola mundo.
```

- ▶ Comentarios (¿para qué?)
- ▶ Directriz del preprocesador (#)
- ▶ Archivos de cabecera/header (.h)
- ▶ Biblioteca estándar (stdin, stdout)
- ▶ Función main (paréntesis)
- ▶ Parámetros y valor de retorno
- ▶ Bloque (llaves)- Cuerpo de la función
- ▶ Parámetros y valor de devolución
- ▶ Enunciados (finaliza con ';'')
- ▶ Caracter de escape ('\')

# Introducción al lenguaje C

## Programas de ejemplo

---

```
1 /* Primer programa en C */
2 #include <stdio.h>
3
4 int main(void)
5 {
6     printf("Hola mundo.\n");
7     return 0;
8 }
```

---

```
Hola mundo.
```

- ▶ Comentarios (¿para qué?)
- ▶ Directriz del preprocesador (#)
- ▶ Archivos de cabecera/header (.h)
- ▶ Biblioteca estándar (stdin, stdout)
- ▶ Función main (paréntesis)
- ▶ Parámetros y valor de retorno
- ▶ Bloque (llaves)- Cuerpo de la función
- ▶ Parámetros y valor de devolución
- ▶ Enunciados (finaliza con ';'')
- ▶ Caracter de escape ('\')
- ▶ Secuencia de escape ('\n')

# Introducción al lenguaje C

## Programas de ejemplo

---

```
1  /* Suma de dos números enteros */
2  #include <stdio.h>
3
4  main()
5  {
6      /* Declaración de variables */
7      int entero1, entero2, suma;
8
9      printf("Ingrese el primer entero: ");
10     scanf("%d", &entero1);
11     printf("Ingrese el segundo entero: ");
12     scanf("%d", &entero2);
13
14     /* Asignación de la variable suma */
15     suma = entero1 + entero2;
16     printf("La suma es: %d\n", suma);
17
18     return 0;      /* finaliza sin error */
19 }
```

---

# Introducción al lenguaje C

## Programas de ejemplo

---

```
1 /* Suma de dos números enteros */
2 #include <stdio.h>
3
4 main()
5 {
6     /* Declaración de variables */
7     int entero1, entero2, suma;
8
9     printf("Ingrese el primer entero: ");
10    scanf("%d", &entero1);
11    printf("Ingrese el segundo entero: ");
12    scanf("%d", &entero2);
13
14    /* Asignación de la variable suma */
15    suma = entero1 + entero2;
16    printf("La suma es: %d\n", suma);
17
18    return 0;    /* finaliza sin error */
19 }
```

---

```
Ingrese el primer entero: 34
Ingrese el segundo entero: 67
La suma es: 101
```

# Introducción al lenguaje C

## Programas de ejemplo

---

```
1 /* Suma de dos números enteros */
2 #include <stdio.h>
3
4 main()
5 {
6     /* Declaración de variables */
7     int entero1, entero2, suma;
8
9     printf("Ingrese el primer entero: ");
10    scanf("%d", &entero1);
11    printf("Ingrese el segundo entero: ");
12    scanf("%d", &entero2);
13
14    /* Asignación de la variable suma */
15    suma = entero1 + entero2;
16    printf("La suma es: %d\n", suma);
17
18    return 0;        /* finaliza sin error */
19 }
```

---

- ▶ Declaración de variables (¿Dónde?, ¿Cuáles son los tipos de datos?)
- ▶ Primer argumento de **printf**: cadena de control de formato
- ▶ **%d**: especificador de conversión
- ▶ **scanf**: cadena de control de formato

# Introducción al lenguaje C

## Programas de ejemplo

---

```
1  /* Suma de dos números enteros */
2  #include <stdio.h>
3
4  main()
5  {
6      /* Declaración de variables */
7      int entero1, entero2;
8
9      printf("Ingrese el primer entero: ");
10     scanf("%d", &entero1);
11     printf("Ingrese el segundo entero: ");
12     scanf("%d", &entero2);
13
14     /* Imprime el resultado */
15     printf("La suma es: %d\n", entero1 + entero2);
16
17     return 0;      /* finaliza sin error */
18 }
```

---

# Introducción al lenguaje C

## Variables

### Variables

Cada variable tiene un *tipo*, un *nombre*, y un *valor*

# Introducción al lenguaje C

## Variables

### Variables

Cada variable tiene un *tipo*, un *nombre*, y un *valor*

### Nombre de variables

Un nombre de variable en C es cualquier **identificador** válido.

Un identificador es una serie de caracteres formados de letras, dígitos y subrayados (-) que no se inicie con un dígito.

C es sensible a las minúsculas y mayúsculas.



# Introducción al lenguaje C

## Variables

### Variables

Cada variable tiene un *tipo*, un *nombre*, y un *valor*

### Nombre de variables

Un nombre de variable en C es cualquier **identificador** válido.

Un identificador es una serie de caracteres formados de letras, dígitos y subrayados (-) que no se inicie con un dígito.

C es sensible a las minúsculas y mayúsculas.

Los **nombres de variables significativos** ayudan a auto-documentar el código.

# Introducción al lenguaje C

## Operadores

### Operadores aritméticos (binarios)

+ , - , / , \* , = , % (módulo - solo con operandos enteros)

# Introducción al lenguaje C

## Operadores

### Operadores aritméticos (binarios)

$+$ ,  $-$ ,  $/$ ,  $*$ ,  $=$ ,  $\%$  (módulo - solo con operandos enteros)

Algunos comentarios:

- ▶ ¿Qué pasa con la división de enteros? (ej.  $17/5$ , y  $17\%5$ )

# Introducción al lenguaje C

## Operadores

### Operadores aritméticos (binarios)

$+$ ,  $-$ ,  $/$ ,  $*$ ,  $=$ ,  $\%$  (módulo - solo con operandos enteros)

Algunos comentarios:

- ▶ ¿Qué pasa con la división de enteros? (ej.  $17/5$ , y  $17\%5$ )
- ▶ Precedencia de operadores

# Introducción al lenguaje C

## Operadores

### Operadores aritméticos (binarios)

$+$ ,  $-$ ,  $/$ ,  $*$ ,  $=$ ,  $\%$  (módulo - solo con operandos enteros)

Algunos comentarios:

- ▶ ¿Qué pasa con la división de enteros? (ej.  $17/5$ , y  $17\%5$ )
- ▶ Precedencia de operadores
  - ▶  $a * (b + c)$

# Introducción al lenguaje C

## Operadores

### Operadores aritméticos (binarios)

`+`, `-`, `/`, `*`, `=`, `%` (módulo - solo con operandos enteros)

Algunos comentarios:

- ▶ ¿Qué pasa con la división de enteros? (ej. `17/5`, y `17%5`)
- ▶ Precedencia de operadores
  - ▶ `a * (b + c)`
  - ▶ `a1 * b1 + a2 * b2`

# Introducción al lenguaje C

## Operadores

### Operadores aritméticos (binarios)

`+`, `-`, `/`, `*`, `=`, `%` (módulo - solo con operandos enteros)

Algunos comentarios:

- ▶ ¿Qué pasa con la división de enteros? (ej. `17/5`, y `17%5`)
- ▶ Precedencia de operadores
  - ▶ `a * (b + c)`
  - ▶ `a1 * b1 + a2 * b2`
  - ▶ `a0 + a1 * x + a2 * x * x`

# Introducción al lenguaje C

## Operadores

### Operadores aritméticos (binarios)

`+`, `-`, `/`, `*`, `=`, `%` (módulo - solo con operandos enteros)

Algunos comentarios:

- ▶ ¿Qué pasa con la división de enteros? (ej. `17/5`, y `17%5`)
- ▶ Precedencia de operadores
  - ▶ `a * (b + c)`
  - ▶ `a1 * b1 + a2 * b2`
  - ▶ `a0 + a1 * x + a2 * x * x`



# Introducción al lenguaje C

## Operadores

### Operadores aritméticos (binarios)

$+$ ,  $-$ ,  $/$ ,  $*$ ,  $=$ ,  $\%$  (módulo - solo con operandos enteros)

Algunos comentarios:

- ▶ ¿Qué pasa con la división de enteros? (ej.  $17/5$ , y  $17\%5$ )
- ▶ Precedencia de operadores
  - ▶  $a * (b + c)$
  - ▶  $a1 * b1 + a2 * b2$
  - ▶  $a0 + a1 * x + a2 * x * x$

Regla de precedencia: primero  $()$ , luego  $*$ ,  $/$ ,  $\%$ , y finalmente  $+$ ,  $-$ .

# Introducción al lenguaje C

## Operadores

### Operadores aritméticos (binarios)

$+$ ,  $-$ ,  $/$ ,  $*$ ,  $=$ ,  $\%$  (módulo - solo con operandos enteros)

Algunos comentarios:

- ▶ ¿Qué pasa con la división de enteros? (ej.  $17/5$ , y  $17\%5$ )
- ▶ Precedencia de operadores
  - ▶  $a * (b + c)$
  - ▶  $a1 * b1 + a2 * b2$
  - ▶  $a0 + a1 * x + a2 * x * x$

Regla de precedencia: primero  $()$ , luego  $*$ ,  $/$ ,  $\%$ , y finalmente  $+$ ,  $-$ .

### Operadores de asignación

$+=$ ,  $-=$ ,  $*=$ ,  $/=$ ,  $\%=$

# Introducción al lenguaje C

## Palabras reservadas

auto	break	case	char
const	continue	default	do
double	else	enum	extern
float	for	goto	if
int	long	register	return
short	signed	sizeof	static
struct	switch	typedef	union
unsigned	void	volatile	while

# Introducción al lenguaje C

## Palabras reservadas

auto	break	case	char
const	continue	default	do
double	else	enum	extern
float	for	goto	if
int	long	register	return
short	signed	sizeof	static
struct	switch	typedef	union
unsigned	void	volatile	while

# Introducción al lenguaje C

## Palabras reservadas

auto	break	case	char
const	continue	default	do
double	else	enum	extern
float	for	goto	if
int	long	register	return
short	signed	sizeof	static
struct	switch	typedef	union
unsigned	void	volatile	while

# Introducción al lenguaje C

## Palabras reservadas

auto	break	case	char
const	continue	default	do
double	else	enum	extern
float	for	goto	if
int	long	register	return
short	signed	sizeof	static
struct	switch	typedef	union
unsigned	void	volatile	while

# Introducción al lenguaje C

## Palabras reservadas

auto	break	case	char
const	continue	default	do
double	else	enum	extern
float	for	goto	if
int	long	register	return
short	signed	sizeof	static
struct	switch	typedef	union
unsigned	void	volatile	while

# Introducción al lenguaje C

## Palabras reservadas

auto	break	case	char
const	continue	default	do
double	else	enum	extern
float	for	goto	if
int	long	register	return
short	signed	sizeof	static
struct	switch	typedef	union
unsigned	void	volatile	while



# Introducción al lenguaje C

## Palabras reservadas

auto	break	case	char
const	continue	default	do
double	else	enum	extern
float	for	goto	if
int	long	register	return
short	signed	sizeof	static
struct	switch	typedef	union
unsigned	void	volatile	while



# Programación estructurada

## Algoritmo, pseudo-código y diagrama de flujo

¿Qué es un algoritmo?

# Programación estructurada

## Algoritmo, pseudo-código y diagrama de flujo

### ¿Qué es un algoritmo?

Un procedimiento para resolver un problema en términos de

- ▶ las acciones a ejecutarse, y
- ▶ el orden en el cual estas acciones deben de ejecutarse

se llama un algoritmo.

# Programación estructurada

## Algoritmo, pseudo-código y diagrama de flujo

### ¿Qué es un algoritmo?

Un procedimiento para resolver un problema en términos de

- ▶ las acciones a ejecutarse, y
- ▶ el orden en el cual estas acciones deben de ejecutarse

se llama un algoritmo.

### ... ¿pseudo-código?

# Programación estructurada

## Algoritmo, pseudo-código y diagrama de flujo

### ¿Qué es un algoritmo?

Un procedimiento para resolver un problema en términos de

- ▶ las acciones a ejecutarse, y
- ▶ el orden en el cual estas acciones deben de ejecutarse

se llama un algoritmo.

### ... ¿pseudo-código?

Es un lenguaje artificial e informal que auxilia a los programadores a desarrollar los algoritmos.

Ayudan al programa “*a pensar*” un programa antes de intentar escribirlo en un lenguaje de programación como C.

El pseudo-código incluye solo enunciados ejecutables.

# Programación estructurada

## Algoritmo, pseudo-código y diagrama de flujo

### ¿Qué es un algoritmo?

Un procedimiento para resolver un problema en términos de

- ▶ las acciones a ejecutarse, y
- ▶ el orden en el cual estas acciones deben de ejecutarse

se llama un algoritmo.

### ... ¿pseudo-código?

Es un lenguaje artificial e informal que auxilia a los programadores a desarrollar los algoritmos.

Ayudan al programa “*a pensar*” un programa antes de intentar escribirlo en un lenguaje de programación como C.

El pseudo-código incluye solo enunciados ejecutables.

### ... ¿y un diagrama de flujo?

# Programación estructurada

## Algoritmo, pseudo-código y diagrama de flujo

### ¿Qué es un algoritmo?

Un procedimiento para resolver un problema en términos de

- ▶ las acciones a ejecutarse, y
- ▶ el orden en el cual estas acciones deben de ejecutarse

se llama un algoritmo.

### ... ¿pseudo-código?

Es un lenguaje artificial e informal que auxilia a los programadores a desarrollar los algoritmos.

Ayudan al programa “*a pensar*” un programa antes de intentar escribirlo en un lenguaje de programación como C.

El pseudo-código incluye solo enunciados ejecutables.

### ... ¿y un diagrama de flujo?

Un diagrama de flujo es una representación gráfica de un algoritmo o de una porción de un algoritmo.



# Programación estructurada

## Estructuras básicas

### Teorema del programa estructurado (Böhm-Jacopini)

# Programación estructurada

## Estructuras básicas

### Teorema del programa estructurado (Böhm-Jacopini)

Establece que toda función computable puede ser implementada en un lenguaje de programación que combine solo tres estructuras lógicas. Esas tres formas (también llamadas estructuras de control) son:

# Programación estructurada

## Estructuras básicas

### Teorema del programa estructurado (Böhm-Jacopini)

Establece que toda función computable puede ser implementada en un lenguaje de programación que combine solo tres estructuras lógicas. Esas tres formas (también llamadas estructuras de control) son:

- ▶ *Secuencia*: ejecución de una instrucción tras otra.

# Programación estructurada

## Estructuras básicas

### Teorema del programa estructurado (Böhm-Jacopini)

Establece que toda función computable puede ser implementada en un lenguaje de programación que combine solo tres estructuras lógicas. Esas tres formas (también llamadas estructuras de control) son:

- ▶ *Secuencia*: ejecución de una instrucción tras otra.
- ▶ *Selección*: ejecución de una de dos instrucciones (o conjuntos), según el valor de una variable booleana.

# Programación estructurada

## Estructuras básicas

### Teorema del programa estructurado (Böhm-Jacopini)

Establece que toda función computable puede ser implementada en un lenguaje de programación que combine solo tres estructuras lógicas. Esas tres formas (también llamadas estructuras de control) son:

- ▶ *Secuencia*: ejecución de una instrucción tras otra.
- ▶ *Selección*: ejecución de una de dos instrucciones (o conjuntos), según el valor de una variable booleana.
- ▶ *Iteración*: ejecución de una instrucción (o conjunto) mientras una variable booleana sea verdadera. Esta estructura lógica también se conoce como ciclo o bucle.

# Programación estructurada

## Estructuras básicas

### Teorema del programa estructurado (Böhm-Jacopini)

Establece que toda función computable puede ser implementada en un lenguaje de programación que combine solo tres estructuras lógicas. Esas tres formas (también llamadas estructuras de control) son:

- ▶ *Secuencia*: ejecución de una instrucción tras otra.
- ▶ *Selección*: ejecución de una de dos instrucciones (o conjuntos), según el valor de una variable booleana.
- ▶ *Iteración*: ejecución de una instrucción (o conjunto) mientras una variable booleana sea verdadera. Esta estructura lógica también se conoce como ciclo o bucle.

En el lenguaje C

- ▶ Secuencia?: Lenguaje secuencial.

# Programación estructurada

## Estructuras básicas

### Teorema del programa estructurado (Böhm-Jacopini)

Establece que toda función computable puede ser implementada en un lenguaje de programación que combine solo tres estructuras lógicas. Esas tres formas (también llamadas estructuras de control) son:

- ▶ *Secuencia*: ejecución de una instrucción tras otra.
- ▶ *Selección*: ejecución de una de dos instrucciones (o conjuntos), según el valor de una variable booleana.
- ▶ *Iteración*: ejecución de una instrucción (o conjunto) mientras una variable booleana sea verdadera. Esta estructura lógica también se conoce como ciclo o bucle.

En el lenguaje C

- ▶ Secuencia?: Lenguaje secuencial.
- ▶ Selección?:

# Programación estructurada

## Estructuras básicas

### Teorema del programa estructurado (Böhm-Jacopini)

Establece que toda función computable puede ser implementada en un lenguaje de programación que combine solo tres estructuras lógicas. Esas tres formas (también llamadas estructuras de control) son:

- ▶ *Secuencia*: ejecución de una instrucción tras otra.
- ▶ *Selección*: ejecución de una de dos instrucciones (o conjuntos), según el valor de una variable booleana.
- ▶ *Iteración*: ejecución de una instrucción (o conjunto) mientras una variable booleana sea verdadera. Esta estructura lógica también se conoce como ciclo o bucle.

En el lenguaje C

- ▶ Secuencia?: Lenguaje secuencial.
- ▶ Selección?: `if`, `if/else`, `switch`.



# Programación estructurada

## Estructuras básicas

### Teorema del programa estructurado (Böhm-Jacopini)

Establece que toda función computable puede ser implementada en un lenguaje de programación que combine solo tres estructuras lógicas. Esas tres formas (también llamadas estructuras de control) son:

- ▶ *Secuencia*: ejecución de una instrucción tras otra.
- ▶ *Selección*: ejecución de una de dos instrucciones (o conjuntos), según el valor de una variable booleana.
- ▶ *Iteración*: ejecución de una instrucción (o conjunto) mientras una variable booleana sea verdadera. Esta estructura lógica también se conoce como ciclo o bucle.

En el lenguaje C

- ▶ Secuencia?: Lenguaje secuencial.
- ▶ Selección?: `if`, `if/else`, `switch`.
- ▶ Iteración/repetición?:

# Programación estructurada

## Estructuras básicas

### Teorema del programa estructurado (Böhm-Jacopini)

Establece que toda función computable puede ser implementada en un lenguaje de programación que combine solo tres estructuras lógicas. Esas tres formas (también llamadas estructuras de control) son:

- ▶ *Secuencia*: ejecución de una instrucción tras otra.
- ▶ *Selección*: ejecución de una de dos instrucciones (o conjuntos), según el valor de una variable booleana.
- ▶ *Iteración*: ejecución de una instrucción (o conjunto) mientras una variable booleana sea verdadera. Esta estructura lógica también se conoce como ciclo o bucle.

En el lenguaje C

- ▶ Secuencia?: Lenguaje secuencial.
- ▶ Selección?: `if`, `if/else`, `switch`.
- ▶ Iteración/repetición?: `while`, `do/while`, `for`.

# Programación estructurada

## Estructuras básicas

### Teorema del programa estructurado (Böhm-Jacopini)

Establece que toda función computable puede ser implementada en un lenguaje de programación que combine solo tres estructuras lógicas. Esas tres formas (también llamadas estructuras de control) son:

- ▶ *Secuencia*: ejecución de una instrucción tras otra.
- ▶ *Selección*: ejecución de una de dos instrucciones (o conjuntos), según el valor de una variable booleana.
- ▶ *Iteración*: ejecución de una instrucción (o conjunto) mientras una variable booleana sea verdadera. Esta estructura lógica también se conoce como ciclo o bucle.

En el lenguaje C

- ▶ Secuencia?: Lenguaje secuencial.
- ▶ Selección?: `if`, `if/else`, `switch`. ¿Qué hacen?
- ▶ Iteración/repetición?: `while`, `do/while`, `for`. ¿Qué hacen?

# Programación estructurada

## Estructuras básicas

### Teorema del programa estructurado (Böhm-Jacopini)

Establece que toda función computable puede ser implementada en un lenguaje de programación que combine solo tres estructuras lógicas. Esas tres formas (también llamadas estructuras de control) son:

- ▶ *Secuencia*: ejecución de una instrucción tras otra.
- ▶ *Selección*: ejecución de una de dos instrucciones (o conjuntos), según el valor de una variable booleana.
- ▶ *Iteración*: ejecución de una instrucción (o conjunto) mientras una variable booleana sea verdadera. Esta estructura lógica también se conoce como ciclo o bucle.

En el lenguaje C

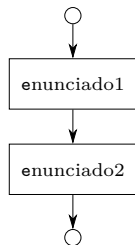
- ▶ Secuencia?: Lenguaje secuencial.
- ▶ Selección?: `if`, `if/else`, `switch`. ¿Qué hacen?
- ▶ Iteración/repetición?: `while`, `do/while`, `for`. ¿Qué hacen?

**C tiene solo 7 estructuras de control**

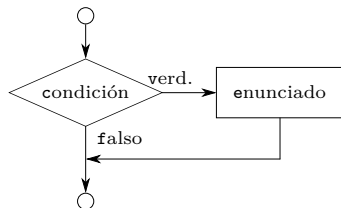
# Programación estructurada

## Estructuras de control

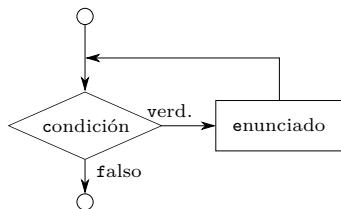
Estructura secuencial



Estructura de selección



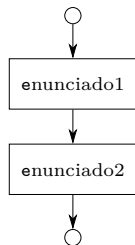
Estructura de repetición



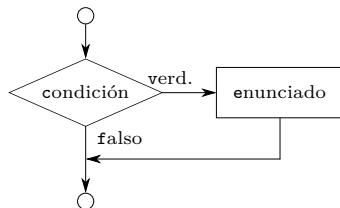
# Programación estructurada

## Estructuras de control

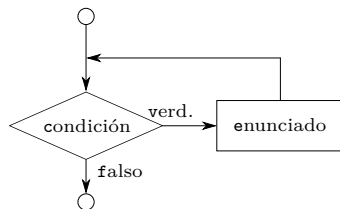
Estructura secuencial



Estructura de selección



Estructura de repetición

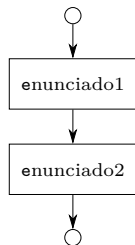


- ▶ Cada estructuras tienen una sola entrada y una sola salida.

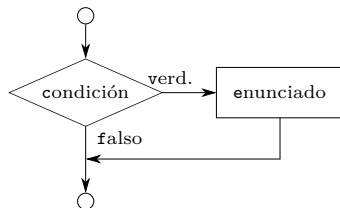
# Programación estructurada

## Estructuras de control

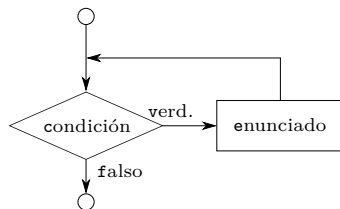
Estructura secuencial



Estructura de selección



Estructura de repetición



- ▶ Cada estructuras tienen una sola entrada y una sola salida.
- ▶ Ellas se puede conectar mediante:
  - ▶ *apilamiento*
  - ▶ *anidamiento*.

# Programación estructurada

## Algunos operadores

### [Condicionales] Operadores de igualdad

==, !=

(menor nivel de precedencia)

### [Condicionales] Operadores relacionales

>, <, >=, <=

(mayor nivel de precedencia)



# Programación estructurada

## Algunos operadores

### [Condicionales] Operadores de igualdad

==, !=

(menor nivel de precedencia)

### [Condicionales] Operadores relacionales

>, <, >=, <=

(mayor nivel de precedencia)

### Incremento y decremento

++, --

(pre/pos incremento/decremento)

# Programación estructurada

## Algunos operadores

### [Condicionales] Operadores de igualdad

`==, !=`

(menor nivel de precedencia)

### [Condicionales] Operadores relacionales

`>, <, >=, <=`

(mayor nivel de precedencia)

### Incremento y decremento

`++, --`

(pre/pos incremento/decremento)

### Operadores lógicos

- ▶ OR lógico: `||`
- ▶ AND lógico: `&&`
- ▶ NOT lógico: `!`



# Estructuras de selección

## La estructura if

### Seudo-código

```
Si calificación es mayor o igual a 60
  Imprimir "Aprobó"
```

### Código C

---

```
if(calificacion >= 60)
  printf("Aprobó\n");
```

---

# Estructuras de selección

## La estructura if

### Seudo-código

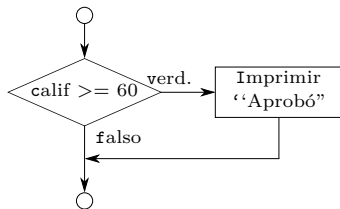
Si calificación es mayor o igual a 60  
Imprimir "Aprobó"

### Código C

---

```
if(calificacion >= 60)
    printf("Aprobó\n");
```

---



# Estructuras de selección

## La estructura if/else

### Seudo-código

```
Si calificación es mayor o igual a 60
  Imprimir "Aprobó"
Si no
  Imprimir "No aprobó"
```

### Código C

---

```
if(calificacion >= 60)
    printf("Aprobó\n");
else
    printf("No aprobó\n");
```

---

# Estructuras de selección

## La estructura if/else

### Seudo-código

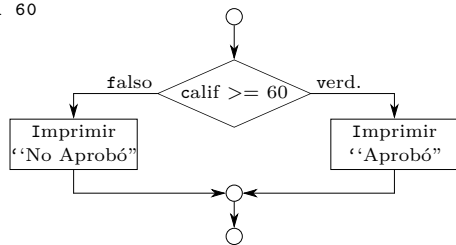
```
Si calificación es mayor o igual a 60
  Imprimir "Aprobó"
Si no
  Imprimir "No aprobó"
```

### Código C

---

```
if(calificacion >= 60)
    printf("Aprobó\n");
else
    printf("No aprobó\n");
```

---



# Estructuras de selección

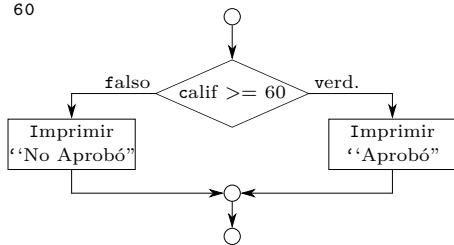
## La estructura if/else

### Seudo-código

```
Si calificación es mayor o igual a 60
  Imprimir "Aprobó"
Si no
  Imprimir "No aprobó"
```

### Código C

```
if(calificacion >= 60)
  printf("Aprobó\n");
else
  printf("No aprobó\n");
```



C tiene el operador condicional '?:' que está relacionado con la estructura if/else.



# Estructuras de selección

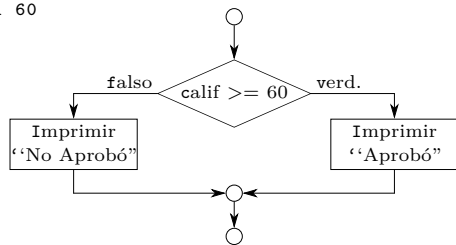
## La estructura if/else

### Seudo-código

```
Si calificación es mayor o igual a 60
  Imprimir "Aprobó"
Si no
  Imprimir "No aprobó"
```

### Código C

```
if(calificacion >= 60)
  printf("Aprobó\n");
else
  printf("No aprobó\n");
```



C tiene el operador condicional '?:' que está relacionado con la estructura if/else.

```
printf("%s\n", calificacion >= 60 ? "Aprobó" : "No aprobó");
```

# Estructuras de selección

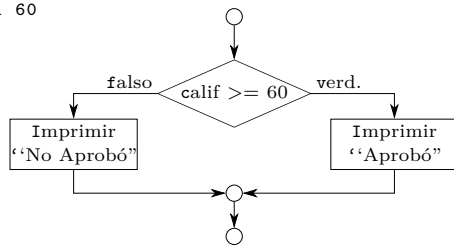
## La estructura if/else

### Seudo-código

```
Si calificación es mayor o igual a 60
  Imprimir "Aprobó"
Si no
  Imprimir "No aprobó"
```

### Código C

```
if(calificacion >= 60)
  printf("Aprobó\n");
else
  printf("No aprobó\n");
```



C tiene el operador condicional '?:' que está relacionado con la estructura if/else.

```
printf("%s\n", calificacion >= 60 ? "Aprobó" : "No aprobó");
```

```
calificacion >= 60 ? printf("Aprobó\n") : printf("No aprobó\n");
```

# Estructuras de selección

## La estructura if/else

Enunciados compuestos:

- ▶ para incluir varios enunciados en el cuerpo de un **if** o **if/else**

# Estructuras de selección

## La estructura if/else

Enunciados compuestos:

- ▶ para incluir varios enunciados en el cuerpo de un **if** o **if/else**

---

```
if(calificacion >= 60)
{
    printf("Aprobó\n");
    printf("Felicitaciones!!\n");
}
else
{
    printf("No aprobó\n");
    printf("Debes recursar\n");
}
```

---

# Estructuras de selección

## La estructura if/else

Enunciados compuestos:

- ▶ para incluir varios enunciados en el cuerpo de un **if** o **if/else**

---

```
if(calificacion >= 60)
{
    printf("Aprobó\n");
    printf("Felicitaciones!!\n");
}
else
{
    printf("No aprobó\n");
    printf("Debes recursar\n");
}
```

---

Algunas preguntas:

- ▶ ¿Qué sucede si no estuvieran las llaves en el **else**?

# Estructuras de selección

## La estructura if/else

Enunciados compuestos:

- ▶ para incluir varios enunciados en el cuerpo de un **if** o **if/else**

---

```
if(calificacion >= 60)
{
    printf("Aprobó\n");
    printf("Felicitaciones!!\n");
}
else
{
    printf("No aprobó\n");
    printf("Debes recursar\n");
}
```

---

Algunas preguntas:

- ▶ ¿Qué sucede si no estuvieran las llaves en el **else**?
- ▶ ¿Qué sucede si se coloca un punto y coma luego de un **if**? ...y el **if/else**?

# Estructuras de selección

## La estructura if/else

Enunciados compuestos:

- ▶ para incluir varios enunciados en el cuerpo de un **if** o **if/else**

---

```
if(calificacion >= 60)
{
    printf("Aprobó\n");
    printf("Felicitaciones!!\n");
}
else
{
    printf("No aprobó\n");
    printf("Debes recursar\n");
}
```

---

Algunas preguntas:

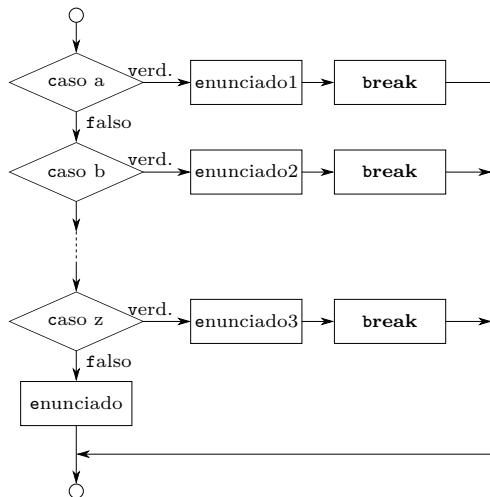
- ▶ ¿Qué sucede si no estuvieran las llaves en el **else**?
- ▶ ¿Qué sucede si se coloca un punto y coma luego de un **if**? ...y el **if/else**?
- ▶ ¿Cómo sería el diagrama de flujo de estructuras **if/else** anidadas?

# Estructuras de selección

## La estructura switch

### Código C

```
switch(caracter)
{
  case 'a':
    enunciado1;
    break;
  case 'b':
    enunciado2;
    break;
  case 'z':
    enunciado3;
    break;
  default:
    enunciado;
}
```



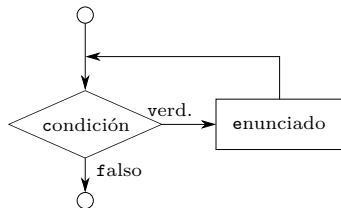


# Estructura de repetición

## La estructuras while

### Código C

```
producto = 2;  
  
while(producto <= 1000)  
    producto = 2*producto;
```

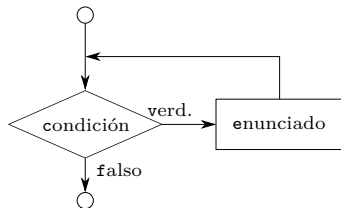


# Estructura de repetición

## La estructuras while

### Código C

```
producto = 2;  
  
while(producto <= 1000)  
    producto = 2*producto;
```



- ▶ ¿Qué sucede si se coloca un punto y coma luego de un **while**?

# Estructura de repetición

## La estructuras do/while

### Código C

---

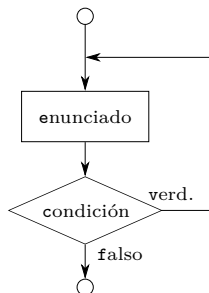
```
/* Imprime del 1 al 10
   utilizando do/while */
#include <stdio.h>

int main(void)
{
    int num = 1;

    do {
        printf("%d\n", num);
    } while(++num <= 10);

    return 0;
}
```

---



# Estructura de repetición

## La estructuras do/while

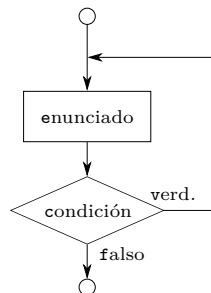
### Código C

```
/* Imprime del 1 al 10
   utilizando do/while */
#include <stdio.h>

int main(void)
{
    int num = 1;

    do {
        printf("%d\n", num);
    } while(++num <= 10);

    return 0;
}
```



### Diferencia entre **while** y **do/while**

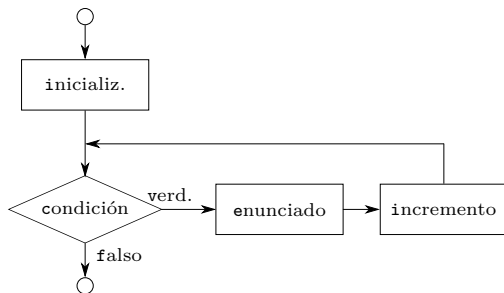
- ▶ En el **while** la condición de continuidad del ciclo se prueba al principio.
- ▶ En el **do/while** la condición de continuidad del ciclo se prueba luego de ejecutar el cuerpo.

# Estructura de repetición

## La estructuras for

### Código C

```
/* Imprime del 1 al 10 utilizando for */  
#include <stdio.h>  
  
int main(void) {  
    int num;  
  
    for(num = 1; num <= 10; num++)  
        printf("%d\n", num);  
  
    return 0;  
}
```



# Estructura de repetición

## La estructuras for

Maneja todas los detalles de la repetición controlada por contador (repetición definida).

### Formato general de la estructura for

---

```
for (expresion1; expresion2; expresion3)  
    enunciado;
```

---

# Estructura de repetición

## La estructuras for

Maneja todas los detalles de la repetición controlada por contador (repetición definida).

### Formato general de la estructura for

---

```
for(expresion1; expresion2; expresion3)
    enunciado;
```

---

### Equivalente con estructura while

---

```
expresion1;
while(expresion2) {
    enunciado;
    expresion3;
}
```

---

# Estructura de repetición

## La estructuras for

---

```
1  /* Sumatoria con estructura for */
2  #include <stdio.h>
3
4  int main(void)
5  {
6      int sum = 0, num;
7
8      for(num = 2; num <= 100; num += 2)
9          sum += num;
10
11     printf("La suma es igual a: %d\n", sum);
12
13     return 0;
14 }
```

---



# Estructura de repetición

## La estructuras for

---

```
1 /* Sumatoria con estructura for */
2 #include <stdio.h>
3
4 int main(void)
5 {
6     int sum = 0, num;
7
8     for(num = 2; num <= 100; num += 2)
9         sum += num;
10
11     printf("La suma es igual a: %d\n", sum);
12
13     return 0;
14 }
```

---

```
La suma es igual a: 2550
```

# Estructura de repetición

## La estructuras for

---

```
1 /* Sumatoria con estructura for */
2 #include <stdio.h>
3
4 int main(void)
5 {
6     int sum = 0, num;
7
8     for(num = 2; num <= 100; sum += num, num += 2)
9         ;
10
11     printf("La suma es igual a: %d\n", sum);
12
13     return 0;
14 }
```

---

```
La suma es igual a: 2550
```



# Funciones

## “Funcionalización” de programas

Desarrollar y mantener un programa grande es más sencillo si está construido a partir de piezas menores o módulos, cada una más fácil de entender y manejar que el programa original

# Funciones

## “Funcionalización” de programas

Desarrollar y mantener un programa grande es más sencillo si está construido a partir de piezas menores o módulos, cada una más fácil de entender y manejar que el programa original → **Modularizar** un programa (diseño descendente).

# Funciones

## “Funcionalización” de programas

Desarrollar y mantener un programa grande es más sencillo si está construido a partir de piezas menores o módulos, cada una más fácil de entender y manejar que el programa original → **Modularizar** un programa (diseño descendente).

Los programas se escriben combinando:

1. funciones que el programador escribe
2. funciones de la *biblioteca estándar* de C

# Funciones

## “Funcionalización” de programas

Desarrollar y mantener un programa grande es más sencillo si está construido a partir de piezas menores o módulos, cada una más fácil de entender y manejar que el programa original → **Modularizar** un programa (diseño descendente).

Los programas se escriben combinando:

1. funciones que el programador escribe
2. funciones de la *biblioteca estándar* de C

Motivos para la funcionalización de un programa:

- ▶ Enfoque (recursivo) de divide y vencerás
- ▶ Reutilización de software
- ▶ Evitar la repetición de código en un programa

# Funciones

## “Funcionalización” de programas

Desarrollar y mantener un programa grande es más sencillo si está construido a partir de piezas menores o módulos, cada una más fácil de entender y manejar que el programa original → **Modularizar** un programa (diseño descendente).

Los programas se escriben combinando:

1. funciones que el programador escribe
2. funciones de la *biblioteca estándar* de C

Motivos para la funcionalización de un programa:

- ▶ Enfoque (recursivo) de divide y vencerás
- ▶ Reutilización de software
- ▶ Evitar la repetición de código en un programa

Las funciones se invocan mediante *llamadas a funciones*.



# Funciones

## “Funcionalización” de programas

Desarrollar y mantener un programa grande es más sencillo si está construido a partir de piezas menores o módulos, cada una más fácil de entender y manejar que el programa original → **Modularizar** un programa (diseño descendente).

Los programas se escriben combinando:

1. funciones que el programador escribe
2. funciones de la *biblioteca estándar* de C

Motivos para la funcionalización de un programa:

- ▶ Enfoque (recursivo) de divide y vencerás
- ▶ Reutilización de software
- ▶ Evitar la repetición de código en un programa

Las funciones se invocan mediante *llamadas a funciones*.

¿Cuál es el tamaño óptimo de una función?

# Funciones

## “Funcionalización” de programas

Desarrollar y mantener un programa grande es más sencillo si está construido a partir de piezas menores o módulos, cada una más fácil de entender y manejar que el programa original → **Modularizar** un programa (diseño descendente).

Los programas se escriben combinando:

1. funciones que el programador escribe
2. funciones de la *biblioteca estándar* de C

Motivos para la funcionalización de un programa:

- ▶ Enfoque (recursivo) de divide y vencerás
- ▶ Reutilización de software
- ▶ Evitar la repetición de código en un programa

Las funciones se invocan mediante *llamadas a funciones*.

¿Cuál es el tamaño óptimo de una función? ¿Y la cantidad de parámetros?

# Funciones

## Prototipo y declaración

### Prototipo de función

```
tipo_de_valor_de_regreso nombre_de_la_funcion(lista de parametros);
```

# Funciones

## Prototipo y declaración

### Prototipo de función

```
tipo_de_valor_de_regreso nombre_de_la_funcion(lista de parametros);
```

### Formato de la declaración

```
tipo_de_valor_de_regreso nombre_de_la_funcion(lista de parametros)  
{  
    definiciones  
  
    instrucciones  
}
```

# Funciones

## Prototipo y declaración

### Prototipo de función

```
tipo_de_valor_de_regreso nombre_de_la_funcion(lista de parametros);
```

### Formato de la declaración

```
tipo_de_valor_de_regreso nombre_de_la_funcion(lista de parametros)
{
    definiciones

    instrucciones
}
```

- ▶ El tipo de regreso por omisión es `int` (depende del estándar)

# Funciones

## Prototipo y declaración

### Prototipo de función

```
tipo_de_valor_de_regreso nombre_de_la_funcion(lista de parametros);
```

### Formato de la declaración

```
tipo_de_valor_de_regreso nombre_de_la_funcion(lista de parametros)
{
    definiciones

    instrucciones
}
```

- ▶ El tipo de regreso por omisión es `int` (depende del estándar)
- ▶ El tipo de regreso `void` significa que no regresa nada

# Funciones

## Prototipo y declaración

### Prototipo de función

```
tipo_de_valor_de_regreso nombre_de_la_funcion(lista de parametros);
```

### Formato de la declaración

```
tipo_de_valor_de_regreso nombre_de_la_funcion(lista de parametros)
{
    definiciones

    instrucciones
}
```

- ▶ El tipo de regreso por omisión es `int` (depende del estándar)
- ▶ El tipo de regreso `void` significa que no regresa nada
- ▶ Las variables declaradas en la definición son locales

# Funciones

## Parámetros

- ▶ La *lista de parámetros* se refiere al tipo, orden y cantidad de parámetros
- ▶ Parámetros:
  - Formales: parámetros en la declaración de la función
  - Verdaderos: parámetros que envía la función llamadora
- ▶ Los parámetros formales son variables locales a la función



# Funciones

## Parámetros

- ▶ La *lista de parámetros* se refiere al tipo, orden y cantidad de parámetros
- ▶ Parámetros:
  - Formales: parámetros en la declaración de la función
  - Verdaderos: parámetros que envía la función llamadora
- ▶ Los parámetros formales son variables locales a la función

## Parámetro vs. argumento (K&R)

- ▶ Parámetro: declaración dentro de los paréntesis seguidos al nombre de la función
- ▶ Argumento: expresión dentro de los paréntesis en una llamada a función

# Funciones

## Parámetros

- ▶ La *lista de parámetros* se refiere al tipo, orden y cantidad de parámetros
- ▶ Parámetros:
  - Formales: parámetros en la declaración de la función
  - Verdaderos: parámetros que envía la función llamadora
- ▶ Los parámetros formales son variables locales a la función

## Parámetro vs. argumento (K&R)

- ▶ Parámetro: declaración dentro de los paréntesis seguidos al nombre de la función
- ▶ Argumento: expresión dentro de los paréntesis en una llamada a función

## Llamada por valor y referencia

**Por valor:** Se hace una copia del valor del argumento. Al modificar la copia no se afecta el valor original.

**Por referencia:** Permite que la función modifique el valor original de la variable.

# Funciones

## Parámetros

- ▶ La *lista de parámetros* se refiere al tipo, orden y cantidad de parámetros
- ▶ Parámetros:
  - Formales: parámetros en la declaración de la función
  - Verdaderos: parámetros que envía la función llamadora
- ▶ Los parámetros formales son variables locales a la función

## Parámetro vs. argumento (K&R)

- ▶ Parámetro: declaración dentro de los paréntesis seguidos al nombre de la función
- ▶ Argumento: expresión dentro de los paréntesis en una llamada a función

## Llamada por valor y referencia

**Por valor:** Se hace una copia del valor del argumento. Al modificar la copia no se afecta el valor original.

**Por referencia:** Permite que la función modifique el valor original de la variable.

En C todas las llamadas son llamadas por valor, la llamada por referencia se simular mediante la utilización de operadores de dirección y de indirección.

# Funciones

## Retorno de una función

- ▶ Si la función no regresa un valor, el control se devuelve a la función llamadora cuando:

# Funciones

## Retorno de una función

- ▶ Si la función no regresa un valor, el control se devuelve a la función llamadora cuando:
  - ▶ se alcanza la llave derecha que termina la función,

# Funciones

## Retorno de una función

- ▶ Si la función no regresa un valor, el control se devuelve a la función llamadora cuando:
  - ▶ se alcanza la llave derecha que termina la función,
  - ▶ o al ejecutar el enunciado `return;`

# Funciones

## Retorno de una función

- ▶ Si la función no regresa un valor, el control se devuelve a la función llamadora cuando:
  - ▶ se alcanza la llave derecha que termina la función,
  - ▶ o al ejecutar el enunciado  
`return;`
- ▶ Si la función regresa un valor, el enunciado  
`return expresion;`  
devuelve el valor *expresion* a la función llamadora.

# Funciones

## Pasaje de arreglos a funciones

### Arreglos

- ▶ Estructura de datos de elementos relacionados del mismo tipo (vs. `struct`)



# Funciones

## Pasaje de arreglos a funciones

### Arreglos

- ▶ Estructura de datos de elementos relacionados del mismo tipo (vs. **struct**)
- ▶ Posiciones de memoria del mismo nombre y mismo tipo.

# Funciones

## Pasaje de arreglos a funciones

### Arreglos

- ▶ Estructura de datos de elementos relacionados del mismo tipo (vs. **struct**)
- ▶ Posiciones de memoria del mismo nombre y mismo tipo.
- ▶ Los arreglos ocupan espacio en memoria. Tipo y cantidad de elementos.

# Funciones

## Pasaje de arreglos a funciones

### Arreglos

- ▶ Estructura de datos de elementos relacionados del mismo tipo (vs. **struct**)
- ▶ Posiciones de memoria del mismo nombre y mismo tipo.
- ▶ Los arreglos ocupan espacio en memoria. Tipo y cantidad de elementos.
- ▶ Los arreglos (y **struct**) son estructuras de datos estáticas.

# Funciones

## Pasaje de arreglos a funciones

### Arreglos

- ▶ Estructura de datos de elementos relacionados del mismo tipo (vs. **struct**)
- ▶ Posiciones de memoria del mismo nombre y mismo tipo.
- ▶ Los arreglos ocupan espacio en memoria. Tipo y cantidad de elementos.
- ▶ Los arreglos (y **struct**) son estructuras de datos estáticas.
- ▶ **#define** para tamaño de arreglos (constante simbólica) → programas dimensionables.

# Funciones

## Pasaje de arreglos a funciones

### Arreglos

- ▶ Estructura de datos de elementos relacionados del mismo tipo (vs. **struct**)
- ▶ Posiciones de memoria del mismo nombre y mismo tipo.
- ▶ Los arreglos ocupan espacio en memoria. Tipo y cantidad de elementos.
- ▶ Los arreglos (y **struct**) son estructuras de datos estáticas.
- ▶ **#define** para tamaño de arreglos (constante simbólica) → programas dimensionables.
- ▶ Arreglos de tipo **char** → cadenas de caracteres.

# Funciones

## Pasaje de arreglos a funciones

### Arreglos

- ▶ Estructura de datos de elementos relacionados del mismo tipo (vs. **struct**)
- ▶ Posiciones de memoria del mismo nombre y mismo tipo.
- ▶ Los arreglos ocupan espacio en memoria. Tipo y cantidad de elementos.
- ▶ Los arreglos (y **struct**) son estructuras de datos estáticas.
- ▶ **#define** para tamaño de arreglos (constante simbólica) → programas dimensionables.
- ▶ Arreglos de tipo **char** → cadenas de caracteres.
- ▶ ¿Cómo se declara?

# Funciones

## Pasaje de arreglos a funciones

### Arreglos

- ▶ Estructura de datos de elementos relacionados del mismo tipo (vs. **struct**)
- ▶ Posiciones de memoria del mismo nombre y mismo tipo.
- ▶ Los arreglos ocupan espacio en memoria. Tipo y cantidad de elementos.
- ▶ Los arreglos (y **struct**) son estructuras de datos estáticas.
- ▶ **#define** para tamaño de arreglos (constante simbólica) → programas dimensionables.
- ▶ Arreglos de tipo **char** → cadenas de caracteres.
- ▶ ¿Cómo se declara? ¿Cómo se inicializan?

# Funciones

## Pasaje de arreglos a funciones

### Arreglos

- ▶ Estructura de datos de elementos relacionados del mismo tipo (vs. **struct**)
- ▶ Posiciones de memoria del mismo nombre y mismo tipo.
- ▶ Los arreglos ocupan espacio en memoria. Tipo y cantidad de elementos.
- ▶ Los arreglos (y **struct**) son estructuras de datos estáticas.
- ▶ **#define** para tamaño de arreglos (constante simbólica) → programas dimensionables.
- ▶ Arreglos de tipo **char** → cadenas de caracteres.
- ▶ ¿Cómo se declara? ¿Cómo se inicializan?
- ▶ El nombre del arreglo es la dirección del primer elemento.



# Funciones

## Pasaje de arreglos a funciones

### Arreglos

- ▶ Estructura de datos de elementos relacionados del mismo tipo (vs. **struct**)
- ▶ Posiciones de memoria del mismo nombre y mismo tipo.
- ▶ Los arreglos ocupan espacio en memoria. Tipo y cantidad de elementos.
- ▶ Los arreglos (y **struct**) son estructuras de datos estáticas.
- ▶ **#define** para tamaño de arreglos (constante simbólica) → programas dimensionables.
- ▶ Arreglos de tipo **char** → cadenas de caracteres.
- ▶ ¿Cómo se declara? ¿Cómo se inicializan?
- ▶ El nombre del arreglo es la dirección del primer elemento.

C pasa los arreglos a las funciones utilizando **llamada por referencia** de forma automática.

# Funciones

## Pasaje de arreglos a funciones

### Función con arreglo

```
double promedio(float datos[], int tam)
{
    . . .
}
```

### y con puntero

```
double promedio(float *datos, int tam)
{
    . . .
}
```

# Funciones

## Pasaje de arreglos a funciones

### Función con arreglo

```
double promedio(float datos[], int tam)
{
    . . .
}
```

### y con puntero

```
double promedio(float *datos, int tam)
{
    . . .
}
```

En la función se puede acceder a los elementos del arreglo utilizando la notación de arreglo o puntero (aritmética de puntero).

# Funciones

## Pasaje de arreglos a funciones

### Función con arreglo

```
double promedio(float datos[], int tam)
{
    . . .
}
```

### y con puntero

```
double promedio(float *datos, int tam)
{
    . . .
}
```

En la función se puede acceder a los elementos del arreglo utilizando la notación de arreglo o puntero (aritmética de puntero).

### Calificador const

```
void imprimir_arreglo(const float datos[], int tam)
{
    . . .
}
```



# Funciones

## Argumentos a la función `main`

```
int main(void) { . . . }
```

# Funciones

## Argumentos a la función `main`

```
int main(void) { . . . }
```

```
int main(int argc, char *argv[]) { . . . }
```

# Funciones

## Argumentos a la función `main`

```
int main(void) { . . . }
```

```
int main(int argc, char *argv[]) { . . . }
```

- ▶ `argc`: cantidad de argumentos en la línea de comandos al ejecutar el programa
- ▶ `argv`: puntero a un vector de cadenas que contiene los argumentos (`argv[argc]` es un puntero `NULL`)