

# Informática II

## Repaso del lenguaje C

Gonzalo F. Perez Paina



Universidad Tecnológica Nacional  
Facultad Regional Córdoba  
UTN-FRC

# 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>BCPL: Basic Combined Programming Language

# 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>BCPL: Basic Combined Programming Language

# Un poco de historia

El primer libro de referencia fue *C Programming Language* (1978) de *Brian Kernighan* y *Dennis Ritchie*



# Un poco de historia

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>1</sup>)

---

<sup>1</sup>American National Standards Institute

# Un poco de historia

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>1</sup>)
- ▶ En 1990 aparece el estándar ISO C (ISO<sup>2</sup>)

---

<sup>1</sup>American National Standards Institute

<sup>2</sup>International Standards Organization

# Un poco de historia

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>1</sup>)
- ▶ En 1990 aparece el estándar ISO C (ISO<sup>2</sup>)
- ▶ En 1999 aparece el estándar C99

---

<sup>1</sup>American National Standards Institute

<sup>2</sup>International Standards Organization

# Un poco de historia

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>1</sup>)
- ▶ En 1990 aparece el estándar ISO C (ISO<sup>2</sup>)
- ▶ En 1999 aparece el estándar C99
- ▶ El último estándar publicado de C es el C11 (ISO/IEC 9899:2011) (IEC<sup>3</sup>)

---

<sup>1</sup>American National Standards Institute

<sup>2</sup>International Standards Organization

<sup>3</sup>International Electrotechnical Commission



# Algunas características

- ▶ Lenguaje de propósitos generales ampliamente utilizado

# Algunas características

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

# 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

# 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

# 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

# Algunos inconvenientes

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

# Algunos inconvenientes

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

# 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>4</sup>

---

<sup>4</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.

# Lenguaje C – Programas de ejemplo

¿Qué contiene en el sig. código fuente?

---

```
1  /*
2  * Ejemplo de programa
3  * en lenguaje C
4  */
5  #include <stdio.h>
6
7  int x = 0;
8
9  int main(void)
10 {
11     int x = 2;
12     {
13         int x = 5;
14     }
15     printf("x = %d\n", x);
16     return x;
17 }
```

---

# Lenguaje C – Programas de ejemplo

¿Qué contiene en el sig. código fuente?

---

```
1  /*
2  * Ejemplo de programa
3  * en lenguaje C
4  */
5  #include <stdio.h>
6
7  int x = 0;
8
9  int main(void)
10 {
11     int x = 2;
12     {
13         int x = 5;
14     }
15     printf("x = %d\n", x);
16     return x;
17 }
```

---

```
x = 2
```

# Lenguaje C – Programas de ejemplo

¿Qué contiene en el sig. código fuente?

## 1. Comentarios (¿para qué?)

---

```
1  /*
2   * Ejemplo de programa
3   * en lenguaje C
4   */
5  #include <stdio.h>
6
7  int x = 0;
8
9  int main(void)
10 {
11     int x = 2;
12     {
13         int x = 5;
14     }
15     printf("x = %d\n", x);
16     return x;
17 }
```

---

```
x = 2
```

# Lenguaje C – Programas de ejemplo

¿Qué contiene en el sig. código fuente?

1. Comentarios (¿para qué?)
2. Directiva del preprocesador (#)

---

```
1  /*
2   * Ejemplo de programa
3   * en lenguaje C
4   */
5  #include <stdio.h>
6
7  int x = 0;
8
9  int main(void)
10 {
11     int x = 2;
12     {
13         int x = 5;
14     }
15     printf("x = %d\n", x);
16     return x;
17 }
```

---

```
x = 2
```



# Lenguaje C – Programas de ejemplo

¿Qué contiene en el sig. código fuente?

---

```
1  /*
2  * Ejemplo de programa
3  * en lenguaje C
4  */
5  #include <stdio.h>
6
7  int x = 0;
8
9  int main(void)
10 {
11     int x = 2;
12     {
13         int x = 5;
14     }
15     printf("x = %d\n", x);
16     return x;
17 }
```

---

1. Comentarios (¿para qué?)
2. Directiva del preprocesador (#)
3. Archivos de cabecera/header (.h)

x = 2

# Lenguaje C – Programas de ejemplo

¿Qué contiene en el sig. código fuente?

---

```
1  /*
2  * Ejemplo de programa
3  * en lenguaje C
4  */
5  #include <stdio.h>
6
7  int x = 0;
8
9  int main(void)
10 {
11     int x = 2;
12     {
13         int x = 5;
14     }
15     printf("x = %d\n", x);
16     return x;
17 }
```

---

1. Comentarios (¿para qué?)
2. Directiva del preprocesador (#)
3. Archivos de cabecera/header (.h)
4. Biblioteca estándar (stdin, stdout)

x = 2

# Lenguaje C – Programas de ejemplo

¿Qué contiene en el sig. código fuente?

---

```
1  /*
2  * Ejemplo de programa
3  * en lenguaje C
4  */
5  #include <stdio.h>
6
7  int x = 0;
8
9  int main(void)
10 {
11     int x = 2;
12     {
13         int x = 5;
14     }
15     printf("x = %d\n", x);
16     return x;
17 }
```

---

1. Comentarios (¿para qué?)
2. Directiva del preprocesador (#)
3. Archivos de cabecera/header (.h)
4. Biblioteca estándar (stdin, stdout)
5. Función main (paréntesis)

x = 2

# Lenguaje C – Programas de ejemplo

¿Qué contiene en el sig. código fuente?

---

```
1  /*
2  * Ejemplo de programa
3  * en lenguaje C
4  */
5  #include <stdio.h>
6
7  int x = 0;
8
9  int main(void)
10 {
11     int x = 2;
12     {
13         int x = 5;
14     }
15     printf("x = %d\n", x);
16     return x;
17 }
```

---

1. Comentarios (¿para qué?)
2. Directiva del preprocesador (#)
3. Archivos de cabecera/header (.h)
4. Biblioteca estándar (stdin, stdout)
5. Función main (paréntesis)
6. Parámetros y valor de retorno

x = 2

# Lenguaje C – Programas de ejemplo

¿Qué contiene en el sig. código fuente?

---

```
1  /*
2  * Ejemplo de programa
3  * en lenguaje C
4  */
5  #include <stdio.h>
6
7  int x = 0;
8
9  int main(void)
10 {
11     int x = 2;
12     {
13         int x = 5;
14     }
15     printf("x = %d\n", x);
16     return x;
17 }
```

---

1. Comentarios (¿para qué?)
2. Directiva del preprocesador (#)
3. Archivos de cabecera/header (.h)
4. Biblioteca estándar (stdin, stdout)
5. Función main (paréntesis)
6. Parámetros y valor de retorno
7. Bloque (llaves) - Cuerpo de la función

x = 2

# Lenguaje C – Programas de ejemplo

¿Qué contiene en el sig. código fuente?

---

```
1  /*
2  * Ejemplo de programa
3  * en lenguaje C
4  */
5  #include <stdio.h>
6
7  int x = 0;
8
9  int main(void)
10 {
11     int x = 2;
12     {
13         int x = 5;
14     }
15     printf("x = %d\n", x);
16     return x;
17 }
```

---

1. Comentarios (¿para qué?)
2. Directiva del preprocesador (#)
3. Archivos de cabecera/header (.h)
4. Biblioteca estándar (stdin, stdout)
5. Función main (paréntesis)
6. Parámetros y valor de retorno
7. Bloque (llaves) - Cuerpo de la función
8. Parámetros y valor de devolución

x = 2

# Lenguaje C – Programas de ejemplo

¿Qué contiene en el sig. código fuente?

---

```
1  /*
2   * Ejemplo de programa
3   * en lenguaje C
4   */
5  #include <stdio.h>
6
7  int x = 0;
8
9  int main(void)
10 {
11     int x = 2;
12     {
13         int x = 5;
14     }
15     printf("x = %d\n", x);
16     return x;
17 }
```

---

1. Comentarios (¿para qué?)
2. Directiva del preprocesador (#)
3. Archivos de cabecera/header (.h)
4. Biblioteca estándar (stdin, stdout)
5. Función main (paréntesis)
6. Parámetros y valor de retorno
7. Bloque (llaves) - Cuerpo de la función
8. Parámetros y valor de devolución
9. Enunciados (finaliza con ';' )

x = 2

# Lenguaje C – Programas de ejemplo

¿Qué contiene en el sig. código fuente?

---

```
1  /*
2  * Ejemplo de programa
3  * en lenguaje C
4  */
5  #include <stdio.h>
6
7  int x = 0;
8
9  int main(void)
10 {
11     int x = 2;
12     {
13         int x = 5;
14     }
15     printf("x = %d\n", x);
16     return x;
17 }
```

---

1. Comentarios (¿para qué?)
2. Directiva del preprocesador (#)
3. Archivos de cabecera/header (.h)
4. Biblioteca estándar (stdin, stdout)
5. Función main (paréntesis)
6. Parámetros y valor de retorno
7. Bloque (llaves) - Cuerpo de la función
8. Parámetros y valor de devolución
9. Enunciados (finaliza con ';' )
10. Caracter de escape ('\')

x = 2



# Lenguaje C – Programas de ejemplo

¿Qué contiene en el sig. código fuente?

---

```
1  /*
2  * Ejemplo de programa
3  * en lenguaje C
4  */
5  #include <stdio.h>
6
7  int x = 0;
8
9  int main(void)
10 {
11     int x = 2;
12     {
13         int x = 5;
14     }
15     printf("x = %d\n", x);
16     return x;
17 }
```

---

x = 2

1. Comentarios (¿para qué?)
2. Directiva del preprocesador (#)
3. Archivos de cabecera/header (.h)
4. Biblioteca estándar (stdin, stdout)
5. Función main (paréntesis)
6. Parámetros y valor de retorno
7. Bloque (llaves) - Cuerpo de la función
8. Parámetros y valor de devolución
9. Enunciados (finaliza con ';' )
10. Caracter de escape ('\')
11. Secuencia de escape ('\n')

# Lenguaje C – Programas de ejemplo

---

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

---

# Lenguaje C – Programas de ejemplo

---

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

---

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

# Lenguaje C – Programas de ejemplo

---

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

---

- ▶ Declaración de variables
  - ▶ ¿Dónde se declaran?
  - ▶ ¿Cuáles son los tipos de datos?

# Lenguaje C – Programas de ejemplo

---

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

---

- ▶ Primer argumento de `printf`: cadena de control de formato
- ▶ `%d`: especificador de conversión
- ▶ `scanf`: cadena de control de formato

# Lenguaje C – Programas de ejemplo

---

```
1  /* Suma de dos números enteros */
2  #include <stdio.h>
3
4  int main(void)
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 }
```

---

¿Qué diferencias hay?

# Variables

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

# 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.



# 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.

# Operadores

## Operadores aritméticos (binarios)

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

# Operadores

## Operadores aritméticos (binarios)

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

Algunos comentarios:

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

# Operadores

## Operadores aritméticos (binarios)

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

Algunos comentarios:

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

# Operadores

## Operadores aritméticos (binarios)

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

Algunos comentarios:

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

# Operadores

## Operadores aritméticos (binarios)

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

Algunos comentarios:

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

# Operadores

## Operadores aritméticos (binarios)

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

Algunos comentarios:

- ▶ ¿Qué pasa con la división de enteros? (p.e.  $17/5$ , 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

## Operadores aritméticos (binarios)

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

Algunos comentarios:

- ▶ ¿Qué pasa con la división de enteros? (p.e.  $17/5$ , 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

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



# Palabras reservadas (32)

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

# Palabras reservadas (32)

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

# Palabras reservadas (32)

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

# Palabras reservadas (32)

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

# Palabras reservadas (32)

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

# Palabras reservadas (32)

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

# Palabras reservadas (32)

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

# Palabras reservadas (32)

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



# Palabras reservadas (32)

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

¿Qué es un algoritmo?

# Programación estructurada

## ¿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.

# Programación estructurada

## ¿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.

## ... ¿pseudo-código?

# Programación estructurada

## ¿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.

## ... ¿pseudo-código?

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

# Programación estructurada

## ¿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.

## ... ¿pseudo-código?

- ▶ Es un lenguaje artificial e informal que auxilia a los programadores a desarrollar los algoritmos.
- ▶ Ayudan al programador “*a pensar*” un programa antes de intentar escribirlo en un lenguaje de programación como C.

# Programación estructurada

## ¿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.

## ... ¿pseudo-código?

- ▶ Es un lenguaje artificial e informal que auxilia a los programadores a desarrollar los algoritmos.
- ▶ Ayudan al programador “*a pensar*” un programa antes de intentar escribirlo en un lenguaje de programación como C.
- ▶ El pseudo-código incluye solo enunciados ejecutables.



# Programación estructurada

## ¿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.

## ... ¿pseudo-código?

- ▶ Es un lenguaje artificial e informal que auxilia a los programadores a desarrollar los algoritmos.
- ▶ Ayudan al programador “*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

## ¿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.

## ... ¿pseudo-código?

- ▶ Es un lenguaje artificial e informal que auxilia a los programadores a desarrollar los algoritmos.
- ▶ Ayudan al programador “*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. Se conoce como ciclo o bucle.

En el lenguaje C

- ▶ Secuencia?:

# 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. 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. 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. 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. Se conoce como ciclo o bucle.

En el lenguaje C

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

# 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. 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. Se conoce como ciclo o bucle.

En el lenguaje C

- ▶ Secuencia?: Lenguaje secuencial.
- ▶ Selección?: `if`, `if/else`, `switch`. ¿Qué hacen?
- ▶ Iteración/repeticion?: `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. Se conoce como ciclo o bucle.

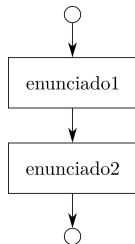
En el lenguaje C

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

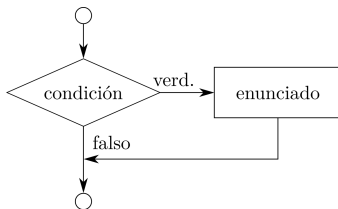
**C tiene solo 7 estructuras de control**

# Programación estructurada – Estructuras básicas

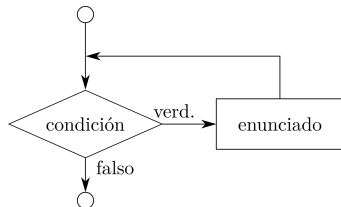
Estructura secuencial



Estructura de selección

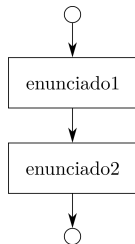


Estructura de repetición

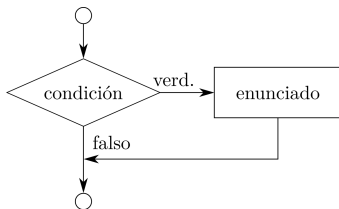


# Programación estructurada – Estructuras básicas

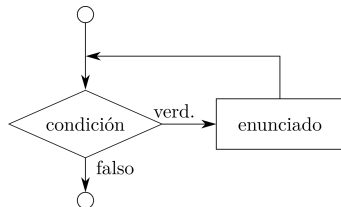
Estructura secuencial



Estructura de selección



Estructura de repetición

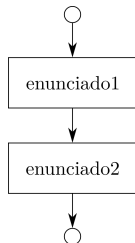


- Cada estructura tiene una sola entrada y una sola salida.

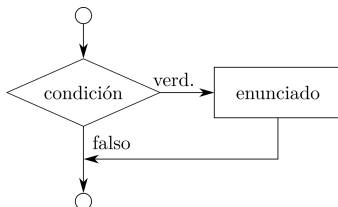


# Programación estructurada – Estructuras básicas

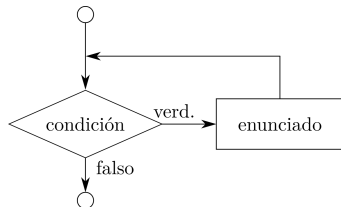
Estructura secuencial



Estructura de selección



Estructura de repetición



- ▶ Cada estructura tiene 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 – 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 – estructura if

## Seudo-código

---

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

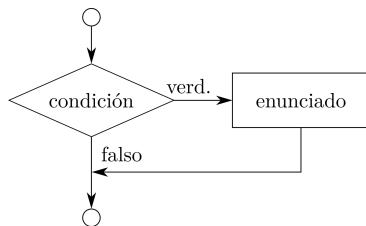
---

## Código C

---

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

---



# Estructuras de selección – 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 – estructura if/else

## Seudo-código

---

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

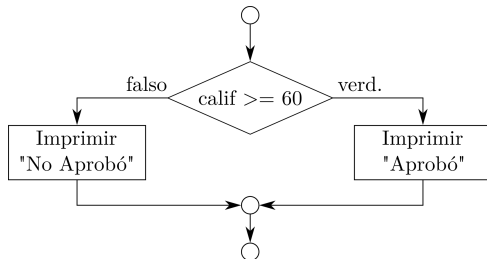
---

## Código C

---

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

---



# Estructuras de selección – estructura if/else

## Seudo-código

---

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

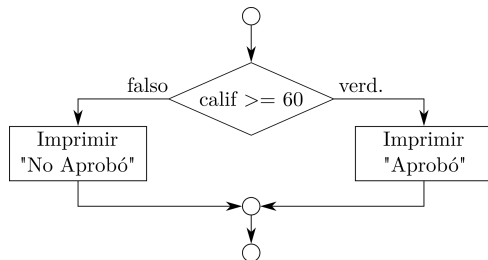
---

## 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 – estructura if/else

## Seudo-código

---

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

---

## Código C

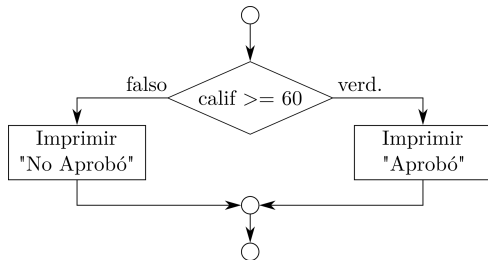
---

```
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 – estructura if/else

## Seudo-código

---

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

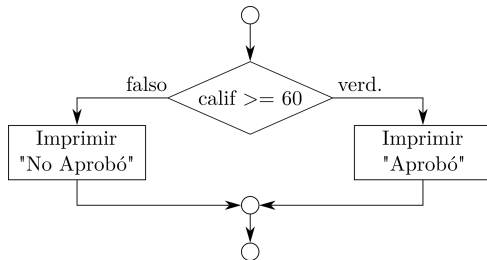
---

## 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 – estructura if/else

Enunciados compuestos:

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

# Estructuras de selección – 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 – 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 recurrar :( \n");
}
```

---

Algunas preguntas:

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

# Estructuras de selección – 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 recurrar :( \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 – 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 recurrar :( \n");
}
```

---

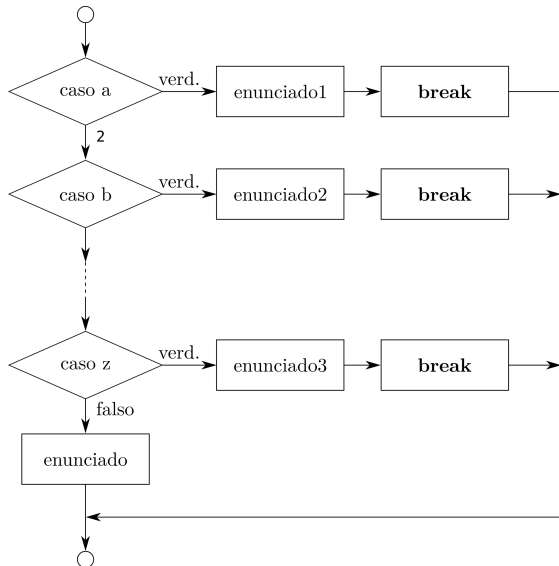
Algunas preguntas:

- ▶ ¿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 – estructura switch

## Código C

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



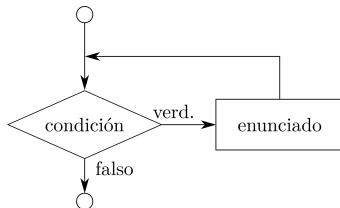
# Estructuras de repetición – estructura while

## Código C

---

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

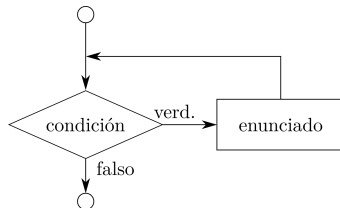
---



# Estructuras de repetición – estructura while

## Código C

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



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

# Estructuras de repetición – estructuras do/while

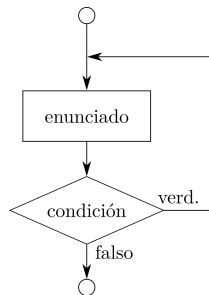
## Código C

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

int main(void)
{
    int num = 1;

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

    return 0;
}
```



# Estructuras de repetición – estructuras do/while

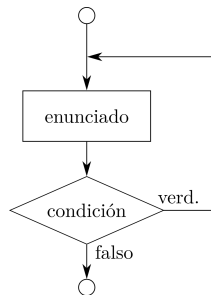
## Código C

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

int main(void)
{
    int num = 1;

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

    return 0;
}
```



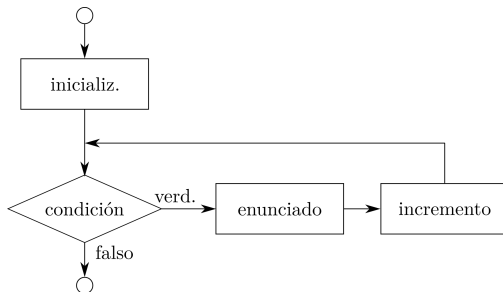
## Diferencia entre while y do/while

- ▶ 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.

# Estructuras de repetición – 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 ", num);  
  
    return 0;  
}
```



# Estructuras de repetición – estructuras for

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

Formato general de la estructura for

---

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

---



# Estructuras de repetición – estructuras for

Maneja todos 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;
}
```

---

# Estructuras de repetición – 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 }
```

---

# Estructuras de repetición – 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
```

# Estructuras de repetición – 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.

# 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.

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.

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.

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.

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.

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

## 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

## 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?

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).

