La ciencia de programar

Jonatan Goméz Perdomo, Ph.D.

Arles Rodríguez, M.Sc.

Camilo Cubides, Ph.D.(c)

Índice general

Índice de tablas

Índice de figuras

Capítulo 1

Introducción

1.1. Lenguaje

Un lenguaje es un conjunto de elementos que nos permite expresarnos y comunicarnos con otros entes, ya sean personas, animales, computadores, etc. Un lenguaje está definido por tres elementos, el léxico, la sintaxis y la semántica.

1.1.1. Léxico

El léxico de un lenguaje lo conforman las unidades mínimas con significado completo. A cada uno de estas unidades mínimas con significado se le conoce como *lexema*[1]. Por ejemplo, en el español, las palabras y los símbolos de puntuación (que son usados para formar frases, oraciones y párrafos) conforman el léxico. A tales lexemas se les asocia un significado preciso en términos de las frases construidas con ellos.

1.1.2. Sintaxis

La sintaxis de un lenguaje explica la forma en que se pueden construir frases en el lenguaje a partir del léxico. Usualmente la sintaxis se presenta como una colección de reglas de reescritura que se definen con una gramática. Éstas son reglas que indican como unos símbolos de la gramática pueden ser reescritos por otros símbolos de la gramática o por lexemas. La idea es que al final del proceso de reescritura sólo se tengan lexemas. Por ejemplo en español una *frase* se puede reescribir como un *sujeto* y un *predicado*, a su vez un *sujeto* se puede reescribir como un *artículo*, un *sustantivo* y un *adjetivo*, finalmente un *sustantivo* puede ser reescrito como la palabra *perro*.

[1]La palabra lexema usada en este libro tiene un significado similar (pero no igual) a la que se usa en lingüística. En lingüística las palabras móvil y móviles se derivan del mismo lexema (móvil), es decir, son el mismo lexema (por las relaciones semánticas propias del español), solamente que tienen diferente gramema (ε, *-es*).

1.1.3. Semántica

La semántica de un lenguaje define la forma en que se le asocia significado (sentido) a las frases construidas mediante la gramática. En español la semántica no es fácil de definir ya que intervienen elementos muy elaborados que han sido construidos de manera natural a través del tiempo (cada objeto/idea conocido(a) por el ser humano esta asociado(a) con una palabra). El sentido de una frase o una oración en español depende mucho del contexto en el que se escribe o dice la frase y del posible conjunto de significados el cual es muy grande. Este hecho es lo que hace difícil, para los computadores actuales, trabajar directamente en lenguaje natural.

1.2. Lenguajes de Programación

Los computadores hacen exactamente lo que se les dice. En programación nosotros tenemos un lenguaje bien definido donde los significados de las frases son exactas. Esto exige que el programador exprese de forma precisa lo que desea hacer.

El lenguaje español es muy ambiguo. Para el computador no hay puntos intermedios, sólo valores de verdad, ceros y unos (verdadero o falso). Desde este punto, la lógica nos permite entender los lenguajes de programación. Los lenguajes de programación son aquellos lenguajes que nos permiten comunicarnos con el computador para ordenarles que hacer.

Al principio programar era muy complicado. En el principio los programas se hacían casi que en *hardware*: se requería que los programas se escribieran cableando ciertas compuertas de la máquina para determinar que el programa hiciera lo que tenía que hacer. El problema consistía en la forma en la que se cableaban los circuitos. Un error en el cableado en este sentido era difícil de detectar.

Posteriormente se pensó en separar el programa de la parte física y así es como llegaron las tarjetas perforadas inspiradas en el invento del telar. Los programas eran representados por huecos en tarjetas. La máquina realizaba lecturas de aquellos huecos de las tarjetas en un orden específico, de desordenarse las tarjetas el programa dejaría de funcionar.

Posteriormente el hombre construyó máquinas de cálculo para tareas muy específicas como investigación y militares, usando dispositivos electro-mecánicos como relés y tubos de vacío. Se programaba revisando las salidas de los estados de los tubos (encendido ó 1 y apagado ó 0). A estos computadores solían acercarseles insectos en busca de calor dañando los tubos. De allí proviene el termino "*bug*" (bicho de programación) conocido actualmente en programación como un defecto en el programa.

Estos computadores dieron paso a los elementos transistorizados. Las máquinas de cómputo de esta generación tenían pocas facilidades de programación. La comunicación se establecía en lenguaje de máquina (lenguaje binario: ceros y unos). Estos aparatos eran grandes y costosos.

Lenguaje de máquina: Es el único lenguaje que entiende el *hardware* (máquina) y usa exclusivamente el sistema binario (ceros y unos). Este lenguaje es específico para cada *hardware* (procesador, dispositivos, etc.).

El programa (tanto códigos de instrucción como datos) es almacenado en memoria. La estructura de una instrucción en lenguaje máquina es la siguiente:

```
CODIGO ARGUMENTO(S)
  0010 00011010
  1010 10111000
  0110 11010001
```

Lenguaje ensamblador: Surgió la necesidad de desarrollar un lenguaje de nivel mayor al de la máquina, y se desarrolló una forma de construir un lenguaje intermedio que empleara mnemónicos (palabras cortas escritas con caracteres alfanuméricos), para codificar las operaciones. Los datos y/o direcciones son codificados generalmente como números en un sistema hexadecimal. Generalmente es específico (aunque no único) para cada lenguaje de máquina. La estructura de una instrucción en este lenguaje es la siguiente:

```
MNEMONICO ARGUMENTO(S)
    ADD R1, F4
    MOV F4, C2
    SUB AX, AX
    MOV AX, 18D
    SUB AX, 18D
```

Un Ensamblador es un software, generalmente escrito en lenguaje de máquina, que es capaz de traducir de lenguaje ensamblador a lenguaje de máquina.

Este lenguaje dio el salto fundamental. Dicho salto se da cuando se logra separar el programa de la máquina empleando los conceptos de máquina de Turing y la arquitectura de Von Neumann. Almacenando el programa en memoria y empleando el *hardware* como elemento de control.

Esto dio origen a los sistemas operativos, logrando que la máquina completa pudiera controlar otro programa.

Lenguajes de alto nivel: Posteriormente se planteó la idea de generar un lenguaje mas parecido al lenguaje natural y que realizara la compilación del programa y generara un programa de código de máquina. Lenguajes como Basic empleaban interpretes tomando cada instrucción y traduciéndola a Ensamblador y de Ensamblador a código de máquina.

Se planteó la idea de tomar un código y traducirlo completamente a lenguaje de máquina mediante un proceso de compilación. El lenguaje de programación C entra en esta categoría de lenguajes.

Dichos lenguajes están basados en una estructura gramatical para codificar estructuras de control y/o instrucciones. Cuenta con un conjunto de palabras reservadas (escritas en lenguaje natural).

Estos lenguajes permiten el uso de símbolos aritméticos y relacionales para describir cálculos matemáticos, y generalmente representan las cantidades numéricas mediante sistema decimal.

Gracias a su estructura gramatical, estos lenguajes permiten al programador olvidar el direccionamiento de memoria (donde cargar datos y/o instrucciones en la memoria), ya que este se realiza mediante el uso de conceptos como el de variable. Los *compiladores* e *interpretes* son software capaz de traducir de un lenguaje de alto nivel al lenguaje ensamblador específico de una máquina. Los primeros toman todo el programa en lenguaje de alto nivel, lo pasan a lenguaje ensamblador y luego lo ejecutan. Los últimos toman instrucción por instrucción, la traducen y la van ejecutando.

Posteriormente se desarrollaron lenguajes intermedios que tomaran una parte compilada y otra interpretada. Es cuando surgen lenguajes como Java y Python. Java compila su código y genera código *bytecode* que se ejecuta en una máquina virtual específica que conoce las instrucciones de *bytecode* permitiendo su ejecución en diferentes sistemas operativos.

Capítulo 2

Lógica Matemática

2.1. Lógica Proposicional

Definición. Una *proposición* es un enunciado del cual se puede decir que es verdadero o falso, pero no ambas cosas simultáneamente. No es necesario saber de antemano sí es verdadero o falso, lo único es que tenga uno sólo de esos valores.

Ejemplos. Los siguientes enunciados son ejemplos de proposiciones

p: $2 + 2 \neq 4$.
q: El jugador está en la casilla $[2, 2]$.
r: El universo es infinito.

Ejemplos. Los siguientes enunciados son ejemplos que no son proposiciones

- ¡No te vayas!.
- ¿Vamos mañana a cine?.
- El lindo perro que corre velozmente por la pradera persiguiendo la pelota.
- No te aprendas la tablas de memoria.

2.1.1. El lenguaje de la lógica proposicional

En la lógica proposicional, el léxico esta definido por tres elementos: las letras proposicionales, los conectivos lógicos y los paréntesis.

Definición. El léxico de la lógica proposicional se compone de tres tipos de lexemas:

símbolos y/o letras proposicionales: $\bot, \top, p, q, r, s, p_0, p_1, \ldots$

conectivos lógicos: $\wedge, \vee, \rightarrow, \leftrightarrow, \neg$

símbolos auxiliares: $(,)$

Las letras proposicionales son usadas para representar proposiciones, por lo tanto el significado de una letra proposicional es el significado que tiene la proposición que dicha letra representa. El símbolo proposicional $\bot$ es usado para representar una proposición con significado siempre *falso*[1], mientras que $\top$ es usado para representar una proposición con significado siempre *verdadero*[2]. Los conectivos lógicos son símbolos que permiten, a partir de símbolos y/o letras proposicionales, formar frases. En la definición mas común de la lógica proposicional clásica, una de las cuales es la lógica clásica estos símbolos son la *negación* ($\neg$), el *o* lógico ($\vee$), el *y* lógico ($\wedge$), la *implicación* ($\rightarrow$) y la *equivalencia* ($\leftrightarrow$). El significado que cada uno de estos conectivos le da a las frases que se construyen con ellos se explicará más adelante[3]. Los paréntesis son usados para agrupar de manera apropiada las frases o fórmulas de la lógica proposicional.

En la lógica proposicional la gramática se describe en términos de *fórmulas bien formadas* (fbf) de manera recursiva[4], es decir, suponiendo que los símbolos y letras proposicionales son fbfs y definiendo nuevas fbfs en términos de fbfs ya construidas.

Definición. La gramática de la lógica proposicional se define recursivamente en términos de fórmulas bien formadas (fbf), así:

i) Si p es un símbolo o letra proposicional, entonces p es una fbf.

ii) Si f es fbf entonces $\neg(f)$ es una fbf.

iii) Si f_1 y f_2 son fbfs entonces: $(f_1 \vee f_2)$, $(f_1 \wedge f_2)$, $(f_1 \rightarrow f_2)$ y $(f_1 \leftrightarrow f_2)$ son fbfs.

En el lenguaje de la lógica proposicional, a diferencia del español u otro lenguaje natural, la semántica es fácil de definir ya que los posibles sentidos que tiene una frase son solamente dos (*verdadero* y *falso*) y las frases que se pueden construir se definen de manera recursiva (fórmulas bien formadas).

Ejemplo. Las siguientes secuencias de símbolos son fórmulas bien formadas:

- f_1: $\big((p \vee \neg(q)) \leftrightarrow (r \wedge s)\big)$
- f_2: $\neg\big((r \rightarrow q) \wedge \neg(q \leftrightarrow s)\big)$

Ejemplo. Las siguientes secuencias de símbolos no son fórmulas bien formadas:

- f_1: $(\wedge\ p)\neg(r \wedge s)\big)$
- f_2: $\big((\vee\ p\ q) \leftrightarrow (q\ p\ \rightarrow)$

Definición. La semántica de la lógica proposicional se define de manera recursiva sobre las fórmulas bien formadas así ($\xi(f)$ se usa para representar el significado de la fórmula bien formada f):

i) Si f es un fbf definida solamente por un símbolo o letra proposicional, el significado de la fórmula f es el mismo significado del símbolo o letra proposicional.

[1] Que se representará abreviadamente por el símbolo F.

[2] Que se representará abreviadamente por el símbolo V.

[3] Existen diversas formas de definir la lógica proposicional clásica dependiendo de los conectivos lógicos usados (símbolo y definición semántica). La presentada aquí es la mas usual y se le dice clásica por la definición semántica de los conectivos lógicos.

[4] En una definición recursiva se definen casos particulares o base y los demás se definen como construcciones sobre estos casos base y sobre estas construcciones.

$\xi(\top)$	$\xi(\bot)$	$\xi(p)$
V	F	significado de la proposición p

ii) Si f es una fbf, entonces:

$\xi(f)$	$\xi\big(\neg(f)\big)$
V	F
F	V

iii) Si f_1 y f_2 son fbfs, entonces:

$\xi(f_1)$	$\xi(f_2)$	$\xi(f_1 \vee f_2)$	$\xi(f_1 \wedge f_2)$	$\xi(f_1 \to f_2)$	$\xi(f_1 \leftrightarrow f_2)$
V	V	V	V	V	V
V	F	V	F	F	F
F	V	V	F	V	F
F	F	F	F	V	V

Ejemplo. Suponga que $\xi(p) = F, \xi(q) = F, \xi(r) = V$, entonces el significado (valor de verdad) de la fórmula bien formada

$$f\colon \neg\big(\big(\neg(p) \to q\big) \wedge \big((r \leftrightarrow q) \vee \neg(\bot)\big)\big)$$

para hallar el significado de f, primero se debe hallar el valor de verdad de los paréntesis más internos y luego con esos resultados ir hallando el valor de verdad de las fórmulas más internas que vayan apareciendo, de esta manera

$\xi(p)$	$\xi(q)$	$\xi(r)$	$\xi\big(\neg(p)\big)$	$\xi\big((r \leftrightarrow q)\big)$	$\xi\big(\neg(\bot)\big)$	$\xi\big(\big(\neg(p) \to q\big)\big)$
F	F	V	V	F	V	F

$\xi\big(\big((r \leftrightarrow q) \vee \neg(\bot)\big)\big)$	$\xi\big(\big(\big(\neg(p) \to q\big) \wedge \big((r \leftrightarrow q) \vee \neg(\bot)\big)\big)\big)$
V	F

$\xi\big(\neg\big(\big(\neg(p) \to q\big) \wedge \big((r \leftrightarrow q) \vee \neg(\bot)\big)\big)\big)$
V

así, $\xi(f) = V$.

2.1.2. Precedencia de conectivos lógicos

Uno de las principales limitaciones de las fórmulas bien formadas es el uso excesivo de los paréntesis, los cuales, en muchos casos, son redundantes. Para evitar este uso excesivo de paréntesis (sin que esto implique que toda fórmula pueda ser escrita sin paréntesis), a los conectores lógicos se les asigna una prioridad que determina de manera exacta el orden en que los paréntesis se deben asumir si no se escriben. Entre más alta es la prioridad de un conector, los paréntesis asociados a él, tienen mayor prelación, es decir, en el proceso

de completar los paréntesis, los paréntesis asociados al operador con más prioridad son adicionados primero que los paréntesis de un conectivo con menor prioridad. Las prioridades asignadas a los operadores se pueden observar el la tabla 2.1. Cuando en la fórmula aparece el mismo operador varias veces y no se puede determinar a cuál se le deben asignar los paréntesis primero, se asignan los paréntesis de izquierda a derecha.

Conectivo	**Prioridad**	**Significado**
$(,)$	1	más alta
$\neg$	2	alta
$\wedge, \vee$	3	media
$\rightarrow, \leftrightarrow$	4	baja

TABLA 2.1. Prioridad de los conectivos lógicos.

Ejemplo. La fórmula $p \rightarrow q \rightarrow r \vee s$ representa la fbf $\big((p \rightarrow q) \rightarrow (r \vee s)\big)$, ya que completando paréntesis:

i) $p \rightarrow q \rightarrow r \vee s$

ii) $p \rightarrow q \rightarrow (r \vee s)$ ($\vee$ prioridad 3)

iii) $(p \rightarrow q) \rightarrow (r \vee s)$ ($\rightarrow$ más a la izquierda prioridad 4)

iv) $\big((p \rightarrow q) \rightarrow (r \vee s)\big)$ ($\rightarrow$ prioridad 4)

2.1.3. Leyes

En la lógica proposicional clásica, una ley lógica es una *equivalencia* o *implicación* entre fórmulas lógicas. Tal equivalencia o implicación lógica debe ser verdadera para cualquier interpretación de las letras proposicionales que conforman las fórmulas relacionadas por la equivalencia (debe ser tautología). Las más famosas leyes lógicas son: *Modus Ponen*, *Modus Tollen*, *Inconsistencia*, *Doble negación*, *Conmutatividad*, *Distributivas*, *Asociativas* y *Morgan*.

2.1.3.1. Interpretación

Dada una colección ζ de símbolos proposicionales, una *interpretación* de ζ es una asignación de valores de verdad a cada una de las letras proposicionales de la colección.

Ejemplo. Sea $\zeta = \{q, r, s\}$.

1. Una interpretación de ζ es: $\xi(q) = V,\ \xi(r) = V,\ \xi(s) = F$.
2. Una interpretación de ζ es: $\xi(q) = F,\ \xi(r) = F,\ \xi(s) = F$.
3. Una interpretación de ζ es: $\xi(q) = F,\ \xi(r) = V,\ \xi(s) = V$.

Propiedad. *Si una colección ζ tiene n letras proposicionales entonces ζ tiene en total 2^n interpretaciones diferentes.*

Ejemplo. Las interpretaciones posibles de la colección de letras proposicionales $\zeta = \{p, q, r\}$, entonces ζ tiene ocho ($2^3 = 8$) interpretaciones:

$\xi(p)$	$\xi(q)$	$\xi(r)$
V	V	V
V	V	F
V	F	V
V	F	F
F	V	V
F	V	F
F	F	V
F	F	F

Nota. *El valor de verdad de una fórmula f para una interpretación I de ζ_f se notará como $\xi_I(f)$.*

2.1.3.2. Tautologías, contradicciones y contingencias

Una fórmula f se dice *tautología* si para cualquier interpretación de su conjunto de letras proposicionales, su significado (valor de verdad) es V, se dice *contradicción* si para cualquier interpretación su significado es F y se dice *contingencia* si no es tautología ni contradicción.

Ejemplo. Determinar el tipo (tautología, contingencia o contradicción) de cada una de las siguientes fórmulas:

1. $f = p \vee q \leftrightarrow q \vee p$
2. $f = p \wedge \neg p$
3. $f = p \wedge (q \vee r)$

Solución.

1. Si $f = p \vee q \leftrightarrow q \vee p$ entonces $\zeta_f = \{p, q\}$

p	q	$p \vee q$	$q \vee p$	$p \vee q \leftrightarrow q \vee p$
V	V	V	V	V
V	F	V	V	V
F	V	V	V	V
F	F	F	F	V

   entonces f es tautología.

2. Si $f = p \wedge \neg p$ entonces $\zeta_f = \{p\}$

p	$\neg p$	$p \wedge \neg p$
V	F	F
F	V	F

entonces f es contradicción.

3. Si $f = p \wedge (q \vee r)$ entonces $\zeta = \{p, q, r\}$

p	q	r	$q \vee r$	$p \wedge (q \vee r)$
V	V	V	V	V
V	V	F	V	V
V	F	V	V	V
V	F	F	F	F
F	V	V	V	F
F	V	F	V	F
F	F	V	V	F
F	F	F	F	F

entonces f es contingencia.

Al esquema de presentar todas las interpretaciones y el valor de verdad de la fórmula se le llama *tabla de verdad* de la fórmula f. Las tablas de verdad son muy útiles para realizar demostraciones a nivel semántico, ya que ellas no solamente se pueden usar con letras proposicionales sino con fórmulas bien formadas, es decir, considerando toda una fórmula bien formada como verdadera o falsa y construyendo la tabla de verdad para dichas fórmulas.

2.1.3.3. Equivalencias Lógicas

Definición. Sean f_1 y f_2 dos fórmulas, se dice que f_1 es *lógicamente equivalente* a f_2, $(f_1 \Leftrightarrow f_2)$ si y solamente si la fórmula $f_1 \leftrightarrow f_2$ es una tautología.

Ejemplo. Las fórmulas $f_1 = \neg(\alpha \wedge \beta)$ y $f_2 = \neg\alpha \vee \neg\beta$ son lógicamente equivalentes, es decir, $\neg(\alpha \wedge \beta) \Leftrightarrow \neg\alpha \vee \neg\beta$, para cualesquiera fórmulas α y β. Para esto, se debe demostrar que $\neg(\alpha \wedge \beta) \leftrightarrow \neg\alpha \vee \neg\beta$ es una tautología; como se aprecia en la siguiente tabla

α	β	$\alpha \wedge \beta$	$\neg(\alpha \wedge \beta)$	$\neg\alpha$	$\neg\beta$	$\neg\alpha \vee \neg\beta$	$\neg(\alpha \wedge \beta) \leftrightarrow \neg\alpha \vee \neg\beta$
V	V	V	F	F	F	F	V
V	F	F	V	F	V	V	V
F	V	F	V	V	F	V	V
F	F	F	V	V	V	V	V

como se observa, $f_1 \leftrightarrow f_2$ es una tautología, por lo tanto, f_1 y f_2 son lógicamente equivalentes.

Las equivalencias lógicas más conocidas se presentan en la tabla 2.2. La demostración de las mismas se deja al lector.

2.1.3.4. Implicaciones Lógicas

Definición. Sea $\Gamma = \{f_1, f_2, \ldots, f_n\}$ una colección de fórmulas (*premisas*) y g una fórmula (*conclusión*), se dice que Γ *implica lógicamente* a g $(\Gamma \Rightarrow g)$, si y solamente si $(f_1 \wedge f_2 \wedge \cdots \wedge f_n) \rightarrow g$ es una tautología.

Equivalencia	**Nombre**
$\alpha \vee \neg\alpha \Leftrightarrow \top$	Tercio excluído
$\alpha \wedge \neg\alpha \Leftrightarrow \bot$	Contradicción
$\alpha \vee \bot \Leftrightarrow \alpha$ $\alpha \wedge \top \Leftrightarrow \alpha$	Identidad
$\alpha \vee \top \Leftrightarrow \top$ $\alpha \wedge \bot \Leftrightarrow \bot$	Dominación
$\alpha \vee \alpha \Leftrightarrow \alpha$ $\alpha \wedge \alpha \Leftrightarrow \alpha$	Idempotencia
$\neg\neg\alpha \Leftrightarrow \alpha$	Doble negación
$\alpha \vee \beta \Leftrightarrow \beta \vee \alpha$ $\alpha \wedge \beta \Leftrightarrow \beta \wedge \alpha$ $\alpha \leftrightarrow \beta \Leftrightarrow \beta \leftrightarrow \alpha$	Conmutativas
$(\alpha \wedge \beta) \wedge \gamma \Leftrightarrow \alpha \wedge (\beta \wedge \gamma)$ $(\alpha \vee \beta) \vee \gamma \Leftrightarrow \alpha \vee (\beta \vee \gamma)$	Asociativas
$\alpha \vee (\beta \wedge \gamma) \Leftrightarrow (\alpha \vee \beta) \wedge (\alpha \vee \gamma)$ $\alpha \wedge (\beta \vee \gamma) \Leftrightarrow (\alpha \wedge \beta) \vee (\alpha \wedge \gamma)$	Distributivas
$\neg(\alpha \wedge \beta) \Leftrightarrow \neg\alpha \vee \neg\beta$ $\neg(\alpha \vee \beta) \Leftrightarrow \neg\alpha \wedge \neg\beta$	Morgan

TABLA 2.2. Equivalencias lógicas.

Ejemplo. Las premisas $\Gamma = \{\neg\beta, \alpha \to \beta\}$ implican lógicamente a $g = \neg\alpha$, para esto es necesario que la fórmula $\big(\neg\beta \wedge (\alpha \to \beta)\big) \to \neg\alpha$ sea una tautología, como se aprecia en la siguiente tabla

α	β	$\neg\beta$	$\alpha \to \beta$	$\neg\beta \wedge (\alpha \to \beta)$	$\neg\alpha$	$\big(\neg\beta \wedge (\alpha \to \beta)\big) \to \neg\alpha$
V	V	F	V	F	F	V
V	F	V	F	F	F	V
F	V	F	V	F	V	V
F	F	V	V	V	V	V

como se observa, $\big(\neg\beta \wedge (\alpha \to \beta)\big) \to \neg\alpha$ es una tautología, por lo tanto, $\Gamma = \{\neg\beta, \alpha \to \beta\}$ implica lógicamente a $g = \neg\alpha$.

Las implicaciones lógicas más conocidas se presentan en la tabla 2.3. Se deja al lector la demostración de las mismas.

2.2. Lógica de predicados

La lógica proposicional no define objetos. Si se tiene la proposición "p: el niño está jugando con la pelota roja y blanca", también se podría hablar de la proposición "q: la foca está jugando con la pelota azul y verde". Al tener frases de este estilo estamos cambiando el sujeto de la oración.

Implicación	Nombre
$\{\alpha, \beta\} \Rightarrow (\alpha \wedge \beta)$	Combinación
$\{\alpha, \beta\} \Rightarrow \alpha$	Ley de simplificación
$\{\alpha, \beta\} \Rightarrow \beta$	Variante de la ley de simplificación
$\{\alpha\} \Rightarrow (\alpha \vee \beta)$	Ley de adición
$\{\beta\} \Rightarrow (\alpha \vee \beta)$	Variante de la adición
$\{\alpha, \alpha \rightarrow \beta\} \Rightarrow \beta$	Modus ponens
$\{\neg\beta, \alpha \rightarrow \beta\} \Rightarrow \neg\alpha$	Modus tollens
$\{\alpha \rightarrow \beta, \beta \rightarrow \gamma\} \Rightarrow (\alpha \rightarrow \gamma)$	Silogismo hipotético
$\{\neg\alpha, \alpha \vee \beta\} \Rightarrow \beta$	Silogismo disyuntivo
$\{\neg\beta, \alpha \vee \beta\} \Rightarrow \alpha$	Variante de silogísmo disyuntivo
$\{\alpha \rightarrow \beta, \neg\alpha \rightarrow \beta\} \Rightarrow \beta$	Ley de casos
$\{\alpha \leftrightarrow \beta\} \Rightarrow (\alpha \rightarrow \beta)$	Eliminación de equivalencia
$\{\alpha \leftrightarrow \beta\} \Rightarrow (\beta \rightarrow \alpha)$	Variante de eliminación de equivalencia
$\{\beta \rightarrow \alpha, \alpha \rightarrow \beta\} \Rightarrow (\alpha \leftrightarrow \beta)$	Introducción de la equivalencia
$\{\alpha, \neg\alpha\} \Rightarrow \beta$	Ley de inconsistencia

TABLA 2.3. Implicaciones lógicas.

En este caso se puede pensar definir sentencias sin un sujeto específico. El sujeto puede cambiar (la foca, el niño). Se realiza referencia a una realidad. Esto da como resultado frases del estilo "x está jugando con y".

x e y son objetos que están relacionados con un predicado y dependiendo del objeto, se tiene una proposición que es verdadera o es falsa. En términos de estos se define un *predicado* como una frase que tiene un valor que verdad dependiendo de los objetos de los que se este hablando. En el ejemplo anterior el predicado es $EstaJugandoCon$ y se escribiría de la forma $EstaJugandoCon(x, y)$, que se interpreta conceptualmente como "x esta jugando con y".

Un predicado da una forma más amplia de hablar. Se podría tener una colección $\{1, 2, 3, 4, 5, 6, 7, 8, 9, 0\}$, y sobre esta colección se define un predicado. Se podría hablar del predicado $esPar(x)$. Si se toma el predicado $esPar(3)$ este tiene un valor de verdad *falso*, si se toma el predicado $esPar(6)$ este tiene un valor de verdad *verdadero*.

A una colección de objetos a los cuales se les desea aplicar el predicado se le llama el *universo de discurso*. Cuando en un predicado se reemplaza una variable x por un valor concreto del universo del discurso, se dice que se está "*instanciando*" la variable x, la fórmula resultante se dice que es una "*instancia*" del predicado inicial. De esta manera cuando se instancian todas la variables de un predicado, lo que se obtiene es una proposición, y por lo tanto cumple la condición de que en ningún caso la instancia puede ser *verdadera* y *falsa* a la vez.

2.2.1. Cuantificadores

Pueden haber predicados como $esDigito(x)$ que para todos los objetos del universo del discurso $\{1, 2, 3, 4, 5, 6, 7, 8, 9, 0\}$ son *verdaderos*. Para este mismo universo, el predi-

cado $esMayora10(x)$ es *falso* para todos los elementos de dicha colección, y el predicado $esModuloAditivo(x)$ es *verdadero* sólo para $x = 0$ y para el resto de los casos será *falso*.

Cuando se desea expresar que un predicado $P(x)$ describe una propiedad sobre todos los elementos del universo del discurso o para sólo algunos, se dice que se está cuantificando la variable x.

Cuando se desea expresar que un predicado describe una propiedad para todos los elementos del universo del discurso, se dice que se está cuantificando *universalmente*. Cuando el predicado describe una propiedad para algunos de los elementos del universo del discurso, se dice que se está cuantificando *existencialmente*.

Para expresar estas nuevas propiedades se necesitan nuevos símbolos, y estos son los símbolos $\forall$ y $\exists$ que permiten ampliar el léxico y se utilizan de la siguiente manera:

- Para notar que una variable x está cuantificada universalmente en un predicado $P(x)$ se utiliza la expresión
$$\forall x P(x)$$
que se lee "*para todo* x $P(x)$".

- Para notar que una variable x está cuantificada existencialmente en un predicado $P(x)$ se utiliza la expresión
$$\exists x P(x)$$
que se lee "*existe* x tal que $P(x)$".

2.2.2. Semántica de los cuantificadores

Cuando en una expresión una variable está cuantificada universalmente, se tiene que

$$\xi\big(\forall x P(x)\big) \Leftrightarrow P(x_1) \wedge P(x_2) \wedge \cdots \wedge P(x_n)$$

para todo valor x_i del universo del discurso.

Cuando en una expresión una variable está cuantificada existencialmente, se tiene que

$$\xi\big(\exists x P(x)\big) \Leftrightarrow P(x_1) \vee P(x_2) \vee \cdots \vee P(x_n)$$

para todo valor x_i del universo del discurso.

Ejemplo. Si se tienen los números dígitos como universo de discurso y se establece como predicado

$$P(x) : x \text{ es múltiplo de } 4,$$

se tiene que $\xi\big(\exists x P(x)\big) = V$ y $\xi\big(\forall x P(x)\big) = F$; esto porque el predicado será cierto cuando se instancia la variable con los valores 0, 4 y 8 $\big(P(0),\ P(4),\ P(8)\big)$, aquí se ha tomado la definición de múltiplo como

> "se dice que un número m es múltiplo de n si existe un entero k, tal que se satisface la igualdad $m = nk$".

2.2.3. Leyes de Morgan para cuantificadores

Con respecto a los cuantificadores, se tienen la siguientes equivalencias que expresan leyes análogas a las leyes de Morgan para la lógica de proposiciones:

$\neg\exists x P(x) \Leftrightarrow \forall x \neg P(x)$**:** no existe un x que cumpla el predicado P quiere decir que para todo x no se cumple el predicado P.

$\neg\forall x P(x) \Leftrightarrow \exists x \neg P(x)$**:** no todo x cumple el predicado P es decir que existe un x que no cumple el predicado.

2.2.4. Particularización universal

Si se piensa por un segundo en el argumento atribuido a Aristóteles,

> "*Todo hombre es Mortal, Aristóteles es un hombre entonces Aristóteles es Mortal*"

Uno de los universos de discurso podría ser

$$U = \text{todas las cosas pensadas por Aristóteles.}$$

En este caso se pueden identificar dos predicados: $Mortal(x)$ y $Humano(x)$.

$$\begin{array}{l} \forall x\big(Humano(x) \rightarrow Mortal(x)\big) \\ Humano(Aristoteles) \\ \hline \therefore Mortal(Aristoteles) \end{array}$$

en este razonamiento se observa que Aristóteles realiza una *particularización universal*, esto consiste en reemplazar una variable que esta cuantificada universalmente por un objeto del universo. Éste es un argumento valido, ya que si se asume la veracidad del predicado $\forall x\big(Humano(x) \rightarrow Mortal(x)\big)$ y también la veracidad de la proposición $Humano(Aristoteles)$, entonces la conclusión $Mortal(Aristoteles)$ debe ser verdadera, ya que si no fuese así, entonces, la proposición $\big(Humano(Aristoteles) \rightarrow Mortal(Aristoteles)\big)$ seria falsa, y eso haría que el predicado $\forall x\big(Humano(x) \rightarrow Mortal(x)\big)$ fuese falso, lo que contradice la suposición de su veracidad.

2.2.5. Lógica de predicados en programación

Cuando un profesor revisa un programa, éste evalúa que para toda entrada dada, se tenga una salida esperada. Si el profesor encuentra un caso para el que el programa no muestra una salida esperada (particularización universal), se concluye que el programa no funciona pues se espera que haga lo que debe hacer para *todos* los posibles casos contemplados.

La lógica de predicados ayuda a establecer precondiciones en la elaboración de los programas. Validaciones de este tipo incluyen verificaciones en los tipos de datos, por ejemplo:

- El cálculo de perímetros y áreas debe funcionar solamente con números positivos.

- El valor de una temperatura requiere que la medición se realice con magnitudes continuas.
- El tiempo promedio de vida de un animal unicelular es una cantidad continua que representa tiempos positivos.

2.3. Ejercicios

1. De los siguientes enunciados ¿cuáles son proposiciones y cuáles no?, justifique su respuesta.

   - Tom Hanks ha ganado dos premios Oscar como mejor actor por dos años consecutivos.
   - Dame una cerveza.
   - Colombia ganó ocho medallas olímpicas en Londres 2012.
   - Todo número primo es impar.
   - $1 + 1 = 2$.
   - La diferencia de dos primos.
   - Todo número par mayor que 2 puede escribirse como suma de dos números primos. (Christian Goldbach, 1742).
   - ¿Que hora es?.
   - $x^n + y^n = z^n$.
   - $x + y = z + y$ si $x = z$.

2. De las siguientes secuencias de símbolos ¿cuáles son fórmulas bien formadas y cuáles no?.

   - $\big((\neg(p) \to r) \wedge (p \neg\ q)\big)$
   - $\big((\neg(p) \leftrightarrow \neg(q)) \leftrightarrow (q \to r)\big)$
   - $(p \wedge q) \vee (q \to p)\big)$
   - $\big((p \leftrightarrow p) \wedge (p \to p) \vee (p \wedge \neg(p))\big)$

3. Escriba la fórmula bien formada que representa cada una de la siguientes secuencias de símbolos:

   - $p \wedge q \leftrightarrow r \vee s \to q$
   - $p \to q \to q \to p$
   - $\neg p \leftrightarrow q \vee \neg r \vee (q \to r) \leftrightarrow \neg\neg q$
   - $p \vee (q \wedge r) \leftrightarrow p \vee q \wedge (p \vee q)$

4. Hallar el significado de cada fórmula que se especifica a continuación con respecto a la interpretación definida para ésta.

   - $f = p \wedge q \leftrightarrow r \vee s \to q$, si $\xi(p) = V$, $\xi(q) = V$, $\xi(r) = V$, $\xi(s) = F$.
   - $f = p \to q \to q \to p$, si $\xi(p) = V$, $\xi(q) = F$.
   - $f = \neg p \leftrightarrow q \vee \neg r \vee (q \to r) \leftrightarrow \neg\neg q$, si $\xi(p) = F$, $\xi(q) = V$, $\xi(r) = V$.
   - $f = p \vee (q \wedge r) \leftrightarrow p \vee q \wedge (p \vee q)$, si $\xi(p) = V$, $\xi(q) = F$, $\xi(r) = V$.

5. Verifique las equivalencias lógicas de la tabla 2.2.

6. Verifique las implicaciones lógicas de la tabla 2.3.

7. Verifique que las fórmulas $f_1 = p \wedge q \vee r$ y $f_2 = p \wedge (q \vee r)$ no son lógicamente equivalentes.

8. Con los operadores lógicos $\neg$ y $\wedge$ es posible expresar los otros operadores lógicos $(\vee, \rightarrow, \leftrightarrow)$ de forma equivalente, de la siguiente manera

$$\begin{aligned} p \vee q &\Leftrightarrow \neg(\neg p \wedge \neg q) \\ p \rightarrow q &\Leftrightarrow \neg(p \wedge \neg q) \\ p \leftrightarrow q &\Leftrightarrow \neg(p \wedge \neg q) \wedge \neg(q \wedge \neg p) \end{aligned}$$

   verificar que efectivamente los operadores lógicos $\vee, \rightarrow, \leftrightarrow$ se pueden expresar en términos de los operadores lógicos $\neg$ y $\wedge$.

9. Con los operadores lógicos $\neg$ y $\vee$ es posible expresar los otros operadores lógicos $(\wedge, \rightarrow, \leftrightarrow)$. Encontrar fórmulas lógicas que contengan sólo los operadores lógicos $\neg$ y $\vee$ que sean equivalentes a las fórmulas $p \wedge q$, $p \rightarrow q$, $p \leftrightarrow q$ y verifique que efectivamente son lógicamente equivalentes.

10. Adicional a los conectivos lógicos presentados, existen otros conectivos, tal como el conectivo *o exclusivo* $(\otimes)$, el cual es muy utilizado en computación, y tiene como objetivo que dadas dos fórmulas f_1 y f_2, la operación $f_1 \otimes f_2$ será únicamente verdadera cuando se cumpla que sólo una de la fórmulas f_1 o f_2 sea verdadera. De esta manera, la semántica para este conectivo es la siguiente

$\xi(f_1)$	$\xi(f_2)$	$\xi(f_1 \otimes f_2)$
V	V	F
V	F	V
F	V	V
F	F	F

   (a) Encuentre una fórmula que sea equivalente lógicamente a la fórmula $f_1 \otimes f_2$, que sólo utilice los operadores lógicos $\neg$ y $\wedge$.

   (b) Encuentre una fórmula que sea equivalente lógicamente a la fórmula $f_1 \otimes f_2$, que sólo utilice los operadores lógicos $\neg$ y $\vee$.

11. Adicional a los conectivos lógicos presentados, existe otro conectivo, tal como el conectivo *barra de Sheffer* $(|)$, para el cual su semántica es la siguiente

$\xi(f_1)$	$\xi(f_2)$	$\xi(f_1 \mid f_2)$
V	V	F
V	F	V
F	V	V
F	F	V

La principal característica de este conectivo es que las fórmulas $\neg f$, $f_1 \vee f_2$, $f_1 \wedge f_2$, $f_1 \to f_2$, $f_1 \leftrightarrow f_2$, son lógicamente equivalentes a fórmulas que sólo contienen el conectivo $|$, como se observa a continuación

$$\begin{aligned} \neg f &\Leftrightarrow f \mid f \\ f_1 \vee f_2 &\Leftrightarrow (f_1 \mid f_1) \mid (f_2 \mid f_2) \\ f_1 \wedge f_2 &\Leftrightarrow (f_1 \mid f_2) \mid (f_1 \mid f_2) \\ f_1 \to f_2 &\Leftrightarrow f_1 \mid (f_2 \mid f_2) \end{aligned}$$

(a) Verifique las anteriores equivalencias lógicas.

(b) Encuentre una fórmula que sólo utilice el conectivo $|$ y que sea lógicamente equivalente a la fórmula $f_1 \leftrightarrow f_2$.

12. Adicional a los conectivos lógicos presentados, existe otro conectivo, tal como el conectivo *flecha de Peirce* ($\downarrow$), para el cual su semántica es la siguiente

$\xi(f_1)$	$\xi(f_2)$	$\xi(f_1 \downarrow f_2)$
V	V	F
V	F	F
F	V	F
F	F	V

La principal característica de este conectivo es que las fórmulas $\neg f$, $f_1 \vee f_2$, $f_1 \wedge f_2$, $f_1 \to f_2$, $f_1 \leftrightarrow f_2$, son lógicamente equivalentes a fórmulas que sólo contienen el conectivo $\downarrow$, como se observa a continuación

$$\begin{aligned} \neg f &\Leftrightarrow f \downarrow f \\ f_1 \vee f_2 &\Leftrightarrow (f_1 \downarrow f_2) \downarrow (f_1 \downarrow f_2) \\ f_1 \wedge f_2 &\Leftrightarrow (f_1 \downarrow f_1) \downarrow (f_2 \downarrow f_2) \\ f_1 \to f_2 &\Leftrightarrow \big((f_1 \downarrow f_1) \downarrow f_2\big) \downarrow \big((f_1 \downarrow f_1) \downarrow f_2\big) \end{aligned}$$

(a) Verifique las anteriores equivalencias lógicas.

(b) Encuentre una fórmula que sólo utilice el conectivo $\downarrow$ y que sea lógicamente equivalente a la fórmula $f_1 \leftrightarrow f_2$.

13. Hallar el valor de verdad para los siguientes predicados con variables cuantificadas y cuyo universo del discurso son los números reales.

i. $\exists x$(x es mayor que cada número natural).

ii. $\exists x$(x es menor que todos los números naturales positivos).

iii. $\forall x$(x tiene inverso aditivo).

iv. $\forall x(x^2 > 0)$.

v. $\forall x(-x \leq 0)$.

vi. $\forall x$(x tiene inverso multiplicativo).

vii. $\exists x(xy = x,$ para todo número real $y)$

viii. $\exists x(x$ es divisor de cada número real).

ix. $\exists x(ax^2 + bx + c = 0,$ para todos los números reales a,b y $c)$.

x. $\exists x(ax^3 + bx^2 + cx + d = 0,$ para todos los números reales a, b, c y $d)$.

14. Usando las leyes de Morgan para cuantificadores, negar los predicados del numeral 13.

Capítulo 3

Teoría de conjuntos

3.1. Conceptos básicos

3.1.1. Conjunto y elemento

Un *conjunto* A es una colección bien definida de objetos. Se dice que una colección A está bien definida si existe un predicado Ψ_A (llamado *constructor del conjunto* A), que determina de manera exacta los objetos que pertenecen a la colección.

Ejemplo. La colección $A = \{1, 2, 3, 4\}$ es un conjunto, ya que el siguiente predicado determina de manera exacta los objetos que pertenecen a A:

$$\Psi_A(x) = \begin{cases} V, & \text{si} \quad x = 1, x = 2, x = 3 \text{ o } x = 4; \\ F, & \text{en otro caso.} \end{cases}$$

Un objeto x se dice que es un *elemento* del conjunto A si y sólo si $\Psi_A(x) = V$. En el caso en que $\Psi_A(x) = F$, se dice que el objeto x *no es un elemento* del conjunto A.

Ejemplo. Para el conjunto $A = \{1, 2, 3, 4\}$, se tiene que 1 es elemento de A ($\Psi_A(1) = V$), y 5 no es elemento de A ($\Psi_A(5) = F$).

Los predicados constructores de conjuntos, permiten definir el siguiente predicado que relaciona elementos con conjuntos

$$\in(x, A) \Leftrightarrow \Psi_A(x)$$

Este predicado es conocido como el predicado *pertenece*.
$x \in A$ es usado para denotar que $\in(x, A)$, y $x \notin A$ es usado para denotar $\neg\big(\in(x, A)\big)$.

3.1.2. Especificación de Conjuntos

Gracias a las definiciones anteriores, un conjunto se puede especificar de dos maneras: por *extensión* y por *comprensión*.

3.1.2.1. Extensión

Si se listan exhaustivamente los elementos que conforman el conjunto o se puede determinar una secuencia que sirve para saber cual es el siguiente de cada elemento en el conjunto, entonces se dice que se está especificando el conjunto por extensión. Esto se hace mediante la siguiente notación

$$A = \{x_1, x_2, \ldots, x_n\}$$

donde x_i son los objetos en el conjunto A. De esta manera el predicado asociado al conjunto es,

$$\Psi_A(x) = \begin{cases} V, & \text{si} \quad x = x_i \quad \text{para algún} \quad i = 1, 2, \ldots, n; \\ F, & \text{en otro caso.} \end{cases}$$

Ejemplos.

1. $A = \{1, 2, 3, 4\}$
2. $B = \{\clubsuit, \blacklozenge, \heartsuit, \spadesuit\}$
3. $C = \{\mathrm{a}, \mathrm{e}, \mathrm{i}, \mathrm{o}, \mathrm{u}\}$
4. $\mathbb{B} = \{V, F\}$
5. $\mathbb{N} = \{0, 1, 2, 3, 4, 5, \ldots\}$
6. $\mathbb{P} = \{1, 2, 3, 4, 5, \ldots\}$
7. $\mathbb{Z} = \{\ldots, -3, -2, -1, 0, 1, 2, 3, \ldots\}$

3.1.2.2. Comprensión

Si se presenta explícitamente el predicado que define el conjunto, se dice que se está especificando el conjunto por comprensión. Esto se hace mediante la siguiente notación:

$$A = \big\{x : \Psi_A(x)\big\}.$$

En esta notación, el símbolo : se lee "*tal que*".

Ejemplos.

- $2\mathbb{N} = \big\{x : (x = 2n) \wedge (n \in \mathbb{N})\big\}$
- $2\mathbb{N} + 1 = \big\{x : (x = 2n + 1) \wedge (n \in \mathbb{N})\big\}$
- $\mathbb{Q} = \big\{x : (x = p/q) \wedge (p, q \in \mathbb{Z}) \wedge (q \neq 0)\big\}$
- $\mathbb{I} = \{x : x \text{ es un número irracional}\}$
- $\mathbb{R} = \{x : x \text{ es un número real}\}$
- $\mathbb{R}^+ = \big\{x : (x \in \mathbb{R}) \wedge (x > 0)\big\}$

- $\mathbb{R}^- = \{x : (x \in \mathbb{R}) \wedge (x < 0)\}$
- $\mathbb{R}^{0,+} = \{x : (x \in \mathbb{R}) \wedge (x \geq 0)\}$ [1]
- $\mathbb{C} = \{x : (x = a + bi) \wedge (a, b \in \mathbb{R}) \wedge (i = \sqrt{-1})\}$
  (i denota la unidad imaginaria)
- $A = \{x : (x \in \mathbb{R}) \wedge (x^2 \leq 1)\}$
- $B = \{x : x \text{ es una pinta del poker}\}$
- $C = \{x : x \text{ es una vocal del idioma español}\}$

El conjunto $\{x : (x \neq x)\}$ es llamado conjunto *vacío* porque no tiene elemento alguno (no existe objeto alguno que sea diferente de si mismo). El símbolo $\varnothing$ es usado para notar al conjunto vacío. Para todo objeto x se tiene que $\Psi_{\varnothing}(x) = (x \in \varnothing) = F$.

3.1.3. Representación de conjuntos mediante diagramas de Venn

Todo conjunto finito no vacío se puede representar mediante los llamados diagramas de Venn, de la siguiente forma

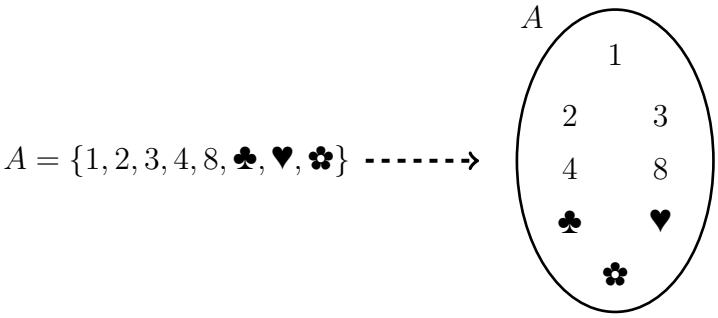


FIGURA 3.1. Representación del conjunto $A = \{1, 2, 3, 4, 8, \clubsuit, \heartsuit, \text{✿}\}$ mediante diagramas de Venn.

3.1.4. Inclusión e igualdad

Sean A y B dos conjuntos, A está *contenido* en B si y sólo si todos los elementos del conjunto A están en el conjunto B. La contenencia entre conjuntos es un predicado que se define de la siguiente manera

$$\text{incluido}(A, B) \Leftrightarrow (x \in A \rightarrow x \in B)$$

Cuando un conjunto A está contenido en un conjunto B, se dice que A es *subconjunto* de B y que B es un *superconjunto* de A.

[1] En el contexto del cálculo o el análisis matemático se usa la notación $\mathbb{R}^*$ para este mismo conjunto, pero en el contexto de la computación, esta notación se utiliza para definir el conjunto de todos los arreglos de tipo real que se estudiarán en el capítulo 10.

$A \subseteq B$ es usado para notar el predicado incluido(A, B), y $A \nsubseteq B$ es usado para denotar $\neg\big(\text{incluido}(A, B)\big)$.

$\varnothing \subseteq A$ y $A \subseteq A$ para todo A conjunto.

Ejemplo. Sean A, B y C los siguientes conjuntos: $A = \{2, 4, 8, \clubsuit, \heartsuit\}$, $B = \{1, 2, 3, 4, \diamondsuit, \spadesuit, \varpropto\}$ y $C = \{0, 2, 4, 6, 8, \clubsuit, \diamondsuit, \heartsuit, \spadesuit\}$, entonces $A \subseteq C$ y $B \nsubseteq C$.

Un conjunto U se dice un *universo* para los conjuntos $A_1, A_2, \ldots, A_n$ si y sólo si $(\forall i = 1, 2, \ldots, n)(A_i \subseteq U)$.

Los universos no son únicos, es decir, se pueden construir diferentes universos para una misma familia de conjuntos.

Ejemplo. Dados los conjuntos $A = \{4, \heartsuit, 2, \clubsuit, 8\}$, $B = \{0, 2, 4, 6, 8, \clubsuit, \diamondsuit, \heartsuit, \spadesuit\}$ y $C = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 0, \clubsuit, \diamondsuit, \heartsuit, \spadesuit, \varpropto\}$, se tiene que B es un universo para A, que C es un universo para A, que C es un universo para B, por lo tanto que C es un universo para A y B.

Sean A y B dos conjuntos, A es *igual* a B si y sólo si los elementos del conjunto A son los mismos del conjunto B. La igualdad entre conjuntos se puede definir mediante el siguiente predicado

$$\text{igual}(A, B) \Leftrightarrow (A \subseteq B) \wedge (B \subseteq A)$$

$A = B$ es usado para notar el predicado igual(A, B).

$$(A = B) \Leftrightarrow (\Psi_A \Leftrightarrow \Psi_B).$$

$A \neq B$ es usado para denotar $\neg\big(\text{igual}(A, B)\big)$.

$A \subsetneq B$ es usado para denotar que $(A \subseteq B) \wedge (A \neq B)$, se dice que A es un *subconjunto propio* de B o que A *está contenido estrictamente* en B.

Ejemplo. Sean A, B y C los siguientes conjuntos: $A = \{2, 4, 8, \clubsuit, \heartsuit\}$, $B = \{4, \heartsuit, 2, \clubsuit, 8\}$ y $C = \{8, \varpropto, 2, \heartsuit, 4, \clubsuit\}$, entonces

- $A \subseteq B$
- $B \subseteq A$
- por lo tanto $A = B$
- $A \subseteq C$
- $C \nsubseteq A$
- es decir $A \neq C$
- de donde $A \subsetneq C$ (A es un subconjunto propio de C)

3.2. Construcción de conjuntos

3.2.1. Unión, intersección y complemento

Sean A y B dos conjuntos, el conjunto *unión* de A y B es el formado con los elementos que están en A o están en B. El conjunto unión es único, se nota $A \cup B$ y se define formalmente mediante el siguiente predicado

$$\Psi_{A\cup B}(x) \Leftrightarrow x \in (A \cup B) := (x \in A) \vee (x \in B)$$

Utilizando notación por comprensión

$$A \cup B = \big\{x : (x \in A) \vee (x \in B)\big\}.$$

En la figura 3.2 se muestra una representación gráfica de la operación $A \cup B$ entre conjuntos.

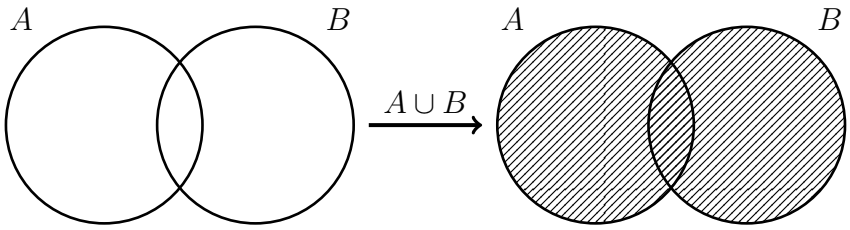


FIGURA 3.2. Representación del conjunto $A \cup B$ mediante diagramas de Venn.

Ejemplo. Sean $A = \{2, 4, 8, \clubsuit, \heartsuit\}$ y $B = \{1, 2, 3, 4, \clubsuit, \heartsuit, \clubsuit\}$, entonces $A \cup B = \{1, 2, 3, 4, 8, \clubsuit, \heartsuit, \clubsuit\}$.

Propiedades. Sean A, B y C conjuntos.

1. $A \cup B = B \cup A$. (conmutatividad)

2. $(A \cup B) \cup C = A \cup (B \cup C)$. (asociatividad)

Sean A y B dos conjuntos, el conjunto *intersección* de A y B es el formado con los elementos que están en A y están en B. El conjunto intersección es único, se nota $A \cap B$ y se define formalmente mediante el siguiente predicado

$$\Psi_{A\cap B}(x) \Leftrightarrow x \in (A \cap B) := (x \in A) \wedge (x \in B)$$

Utilizando notación por comprensión

$$A \cap B = \big\{x : (x \in A) \wedge (x \in B)\big\}.$$

En la figura 3.3 se muestra una representación gráfica de la operación $A \cap B$ entre conjuntos.

Ejemplo. Sean $A = \{2, 4, 8, \clubsuit, \heartsuit\}$ y $B = \{1, 2, 3, 4, \clubsuit, \heartsuit, \clubsuit\}$, entonces $A \cap B = \{2, 4, \clubsuit, \heartsuit\}$.

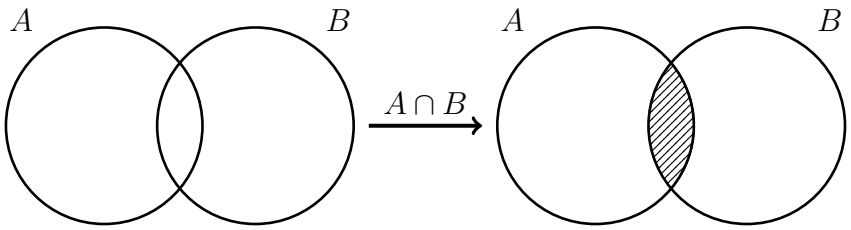


FIGURA 3.3. Representación del conjunto $A \cap B$ mediante diagramas de Venn.

Propiedades. Sean A, B y C conjuntos.

1. $A \cap B = B \cap A$. (conmutatividad)
2. $(A \cap B) \cap C = A \cap (B \cap C)$. (asociatividad)

Sean A y B dos conjuntos, el conjunto *complemento* de A *relativo* a B es el conjunto de todos los elementos que están en B y que no están en A. El conjunto complemento relativo es único, se nota $\overline{A}^B$ y se define formalmente mediante el siguiente predicado

$$\Psi_{\overline{A}^B}(x) \Leftrightarrow x \in \left(\overline{A}^B\right) := (x \notin A) \wedge (x \in B)$$

Utilizando notación por comprensión

$$\overline{A}^B = \big\{x : (x \notin A) \wedge (x \in B)\big\}.$$

En la figura 3.4 se muestra una representación gráfica de la operación $\overline{A}^B$ entre conjuntos.

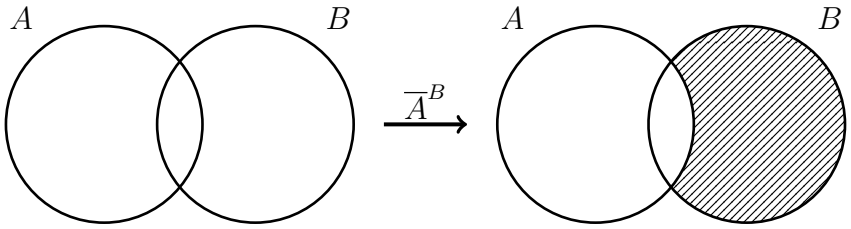


FIGURA 3.4. Representación del conjunto $\overline{A}^B$ mediante diagramas de Venn.

Ejemplo. Sean $A = \{2, 4, 8, \clubsuit, \heartsuit\}$ y $B = \{1, 2, 3, 4, \clubsuit, \heartsuit, \text{✿}\}$, entonces $\overline{A}^B = \{1, 3, \text{✿}\}$ y $\overline{B}^A = \{8\}$.

Cuando B es un universo para A, el conjunto $\overline{A}^B$ es llamado *complemento* de A y es notado $\overline{A}$.

En la figura 3.5 se muestra una representación gráfica de la operación $\overline{A}$ sobre un conjunto.

Ejemplo. Sean $U = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 0, \clubsuit, \diamondsuit, \heartsuit, \spadesuit, \text{✿}\}$ un universo para $A = \{2, 4, 8, \clubsuit, \heartsuit\}$, entonces $\overline{A} = \{1, 3, 5, 6, 7, 9, 0, \diamondsuit, \spadesuit, \text{✿}\}$.

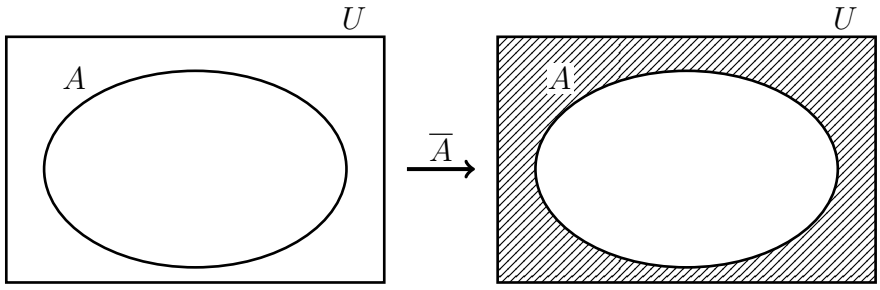


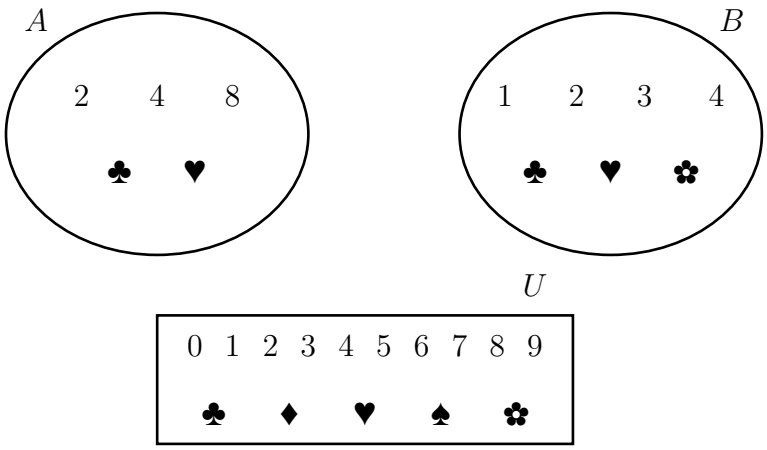
FIGURA 3.5. Representación del conjunto $\overline{A}$ mediante diagramas de Venn.

Propiedades. Sean A y B conjuntos, U un universo para A y B.

1. $A \cup \overline{A} = U$.
2. $A \cap \overline{A} = \varnothing$.
3. $\overline{(A \cap B)} = \overline{A} \cup \overline{B}$.
4. $\overline{(A \cup B)} = \overline{A} \cap \overline{B}$.

3.2.2. Resumen de las operaciones básicas entre conjuntos

Dados los siguientes conjuntos representados usando diagramas de Venn, entonces se van a mostrar algunos ejemplos de la representación de operaciones entre conjuntos usando diagramas de Venn



Ejemplo. Ejemplo de la unión entre conjuntos usando diagramas de Venn $A = \{2, 4, 8, \clubsuit, \heartsuit\}$, $B = \{1, 2, 3, 4, \clubsuit, \heartsuit, \text{✿}\}$

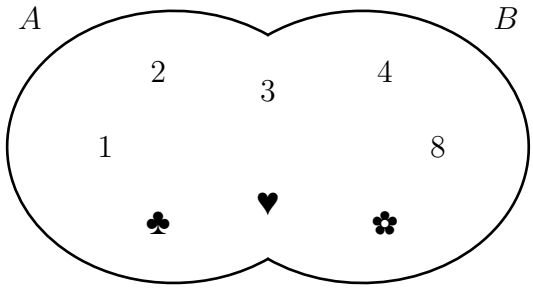


FIGURA 3.6. Representación del conjunto $A \cup B = \{1, 2, 3, 4, 8, \clubsuit, \heartsuit, \text{✿}\}$ mediante diagramas de Venn.

Ejemplo. Ejemplo de la intersección entre conjuntos usando diagramas de Venn $A = \{2, 4, 8, \clubsuit, \heartsuit\}$, $B = \{1, 2, 3, 4, \clubsuit, \heartsuit, \text{✿}\}$

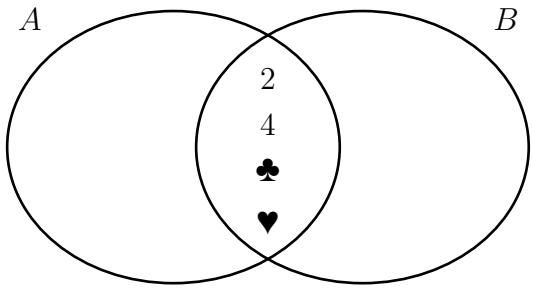


FIGURA 3.7. Representación del conjunto $A \cap B = \{2, 4, \clubsuit, \heartsuit\}$ mediante diagramas de Venn.

Ejemplo. Ejemplo del complemento relativo entre conjuntos usando diagramas de Venn $A = \{2, 4, 8, \clubsuit, \heartsuit\}$, $B = \{1, 2, 3, 4, \clubsuit, \heartsuit, \text{✿}\}$

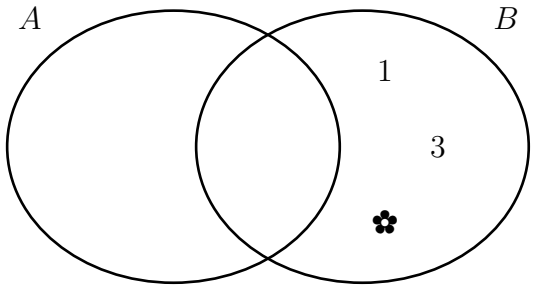


FIGURA 3.8. Representación del conjunto $\overline{A}^B = \{1, 3, \text{✿}\}$ mediante diagramas de Venn.

Ejemplo. Ejemplo del complemento relativo entre conjuntos usando diagramas de Venn $A = \{2, 4, 8, \clubsuit, \heartsuit\}$, $B = \{1, 2, 3, 4, \clubsuit, \heartsuit, \varpropto\}$

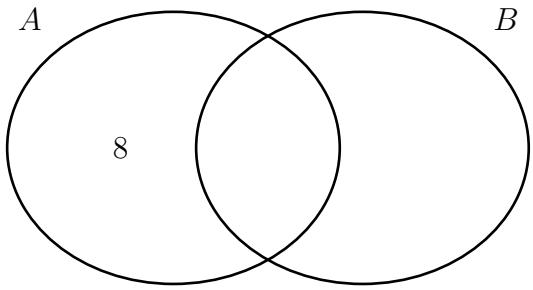


FIGURA 3.9. Representación del conjunto $\overline{B}^A = \{8\}$ mediante diagramas de Venn.

Ejemplo. Ejemplo del complemento de un conjunto usando diagramas de Venn $A = \{2, 4, 8, \clubsuit, \heartsuit\}$, $U = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 0, \clubsuit, \diamondsuit, \heartsuit, \spadesuit, \varpropto\}$

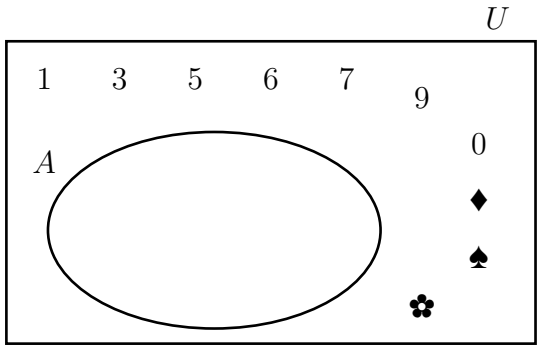
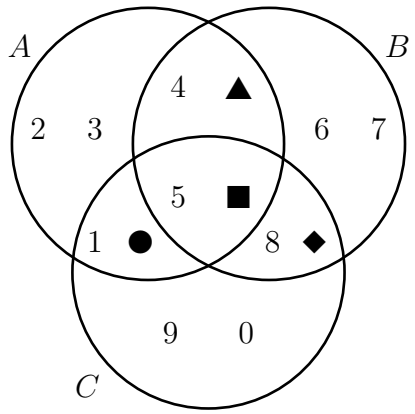


FIGURA 3.10. Representación del conjunto $\overline{A} = \{1, 3, 5, 6, 7, 9, 0, \diamondsuit, \spadesuit, \varpropto\}$ mediante diagramas de Venn.

Ejemplo. Dados los conjuntos $A = \{1, 2, 3, 4, 5, \bullet, \blacksquare, \blacktriangle\}$, $B = \{4, 5, 6, 7, 8, \blacksquare, \blacktriangle, \blacklozenge\}$ y $C = \{0, 1, 5, 8, 9, \bullet, \blacksquare, \blacklozenge\}$, se tiene que el siguiente diagrama presenta todos los posibles casos de contenencia para los conjuntos A, B y C.



3.2.3. Conjunto de partes

Sea A un conjunto, el conjunto *partes* (o *potencia* o *exponencial*) de A es el conjunto de todos los subconjuntos de A. El conjunto de partes es único, se nota $\wp(A)$ y se define formalmente mediante el siguiente predicado:

$$\Psi_{\wp(A)}(X) \Leftrightarrow X \in \big(\wp(A)\big) := (X \subseteq A)$$

Utilizando notación por comprensión

$$\wp(A) = \{X : X \subseteq A\}.$$

Ejemplo. Para el conjunto $A = \{1, 2, 3\}$ se tiene que

$$\wp(A) = \big\{\varnothing, \{1\}, \{2\}, \{3\}, \{1, 2\}, \{1, 3\}, \{2, 3\}, \{1, 2, 3\}\big\}.$$

3.2.4. Producto cartesiano

Para todo par de objetos a y b existe un único objeto c, llamado *pareja ordenada* de a con b y notado $c = (a, b)$, para el cual:

1. $\pi_1(x, y) = \begin{cases} V, & \text{si} \quad x = c \wedge y = a; \\ F, & \text{en otro caso.} \end{cases}$

2. $\pi_2(x, y) = \begin{cases} V, & \text{si} \quad x = c \wedge y = b; \\ F, & \text{en otro caso.} \end{cases}$

Los objetos (a, b) y (b, a) no son iguales cuando $a \neq b$, ya que, $\pi_1\big((a, b), a\big) = V$ y $\pi_1\big((b, a), a\big) = F$.

El conjunto *producto cartesiano* de dos conjuntos A y B, es el conjunto de todas las parejas ordenadas, cuya primer componente es un elemento del conjunto A y cuya segunda componente es un elemento del conjunto B. El conjunto producto cartesiano es único y se nota $A \times B$. El conjunto producto cartesiano puede ser definido mediante el siguiente predicado

$$\Psi_{A\times B}\big((x, y)\big) \Leftrightarrow (x, y) \in A \times B := (x \in A) \wedge (y \in B)$$

Utilizando notación por comprensión

$$A \times B = \big\{(x, y) : (x \in A) \wedge (y \in B)\big\}.$$

Ejemplo. Para los conjuntos $A = \{1, 2, 3\}$ y $B = \{\clubsuit, \blacklozenge, \heartsuit, \spadesuit\}$ se tiene que

$$A \times B = \big\{(1, \clubsuit), (1, \blacklozenge), (1, \heartsuit), (1, \spadesuit), \\ (2, \clubsuit), (2, \blacklozenge), (2, \heartsuit), (2, \spadesuit), (3, \clubsuit), (3, \blacklozenge), (3, \heartsuit), (3, \spadesuit)\big\}.$$

$$B \times A = \big\{(\clubsuit, 1), (\clubsuit, 2), (\clubsuit, 3), (\blacklozenge, 1), (\blacklozenge, 2), (\blacklozenge, 3), \\ (\heartsuit, 1), (\heartsuit, 2), (\heartsuit, 3), (\spadesuit, 1), (\spadesuit, 2), (\spadesuit, 3)\big\}.$$

3.2.5. Producto cartesiano generalizado

Sean $A_1, A_2, \ldots, A_n$, n conjuntos, el producto cartesiano generalizado de dichos conjuntos está definido por las n-tuplas ordenadas $(a_1, a_2, \ldots, a_n)$ con $a_i \in A_i$, $\forall i = 1, 2, 3, \ldots, n$.

Utilizando notación por comprensión

$$A_1 \times A_2 \times \cdots \times A_n = \big\{(a_1, a_2, \ldots, a_n) : (a_i \in A_i), \forall i = 1, 2, 3, \ldots, n\big\}.$$

Ejemplo. Para los conjuntos $A = \{1, 2, 3\}$, $B = \{\clubsuit, \blacklozenge, \heartsuit, \spadesuit\}$ y $C = \{+, *\}$ se tiene que

$$\begin{aligned} A \times B \times C = \big\{&(1, \clubsuit, +), (1, \clubsuit, *), (1, \blacklozenge, +), (1, \blacklozenge, *), (1, \heartsuit, +), (1, \heartsuit, *),\\ &(1, \spadesuit, +), (1, \spadesuit, *), (2, \clubsuit, +), (2, \clubsuit, *), (2, \blacklozenge, +), (2, \blacklozenge, *),\\ &(2, \heartsuit, +), (2, \heartsuit, *), (2, \spadesuit, +), (2, \spadesuit, *), (3, \clubsuit, +), (3, \clubsuit, *),\\ &(3, \blacklozenge, +), (3, \blacklozenge, *), (3, \heartsuit, +), (3, \heartsuit, *), (3, \spadesuit, +), (3, \spadesuit, *)\big\}. \end{aligned}$$

3.2.6. Cardinalidad

Todos los conjuntos poseen una propiedad muy importante llamada *cardinalidad*, ésta se refiere a la cantidad de elementos que posee el conjunto. El cardinal de un conjunto A es único y se denota por $|A|$.

Con base en el concepto de cardinal se puede decidir si un conjunto es *finito* o *infinito*, así, un conjunto es finito si el conjunto es el vacío o su cardinal es un número natural, en caso contrario se dice que es infinito.

Ejemplo. Para los siguientes conjuntos finitos

$$\begin{aligned} A &= \varnothing & F &= \{\clubsuit, \blacklozenge, \heartsuit, \spadesuit\}\\ B &= \{0\} & G &= \{1, 3, 5, 7, 9, \clubsuit, \heartsuit\}\\ C &= \{\maltese\} & H &= \{2, 4, 6, 8, 0, \blacklozenge, \spadesuit\}\\ D &= \{\varnothing\} & I &= \{1, 2, 3, 4, 5, 6, 7, 8, 9, 0, \clubsuit, \blacklozenge, \heartsuit, \spadesuit, \maltese\}\\ E &= \big\{\varnothing, \{\varnothing\}\big\} \end{aligned}$$

se tiene que:

- $|A| = 0$
- $|B| = 1$
- $|C| = 1$
- $|D| = 1$
- $|E| = 2$
- $|F| = 4$
- $|G| = 7$
- $|H| = 7$
- $|I| = 15$

Para el caso de los conjuntos infinitos, estos pueden tener distinto cardinal,

Ejemplo.

- $|\mathbb{N}| = \aleph_0$
  (que se lee "*alef cero*")
- $|\mathbb{P}| = \aleph_0$
- $|2\mathbb{N}| = \aleph_0$
- $|2\mathbb{N} + 1| = \aleph_0$

- $|\mathbb{Z}| = \aleph_0$
- $|\mathbb{Q}| = \aleph_0$
- $|\mathbb{N} \times \mathbb{N}| = \aleph_0$
- $|\wp(\mathbb{N})| = \aleph_1 = 2^{\aleph_0}$
- $|\mathbb{I}| = \aleph_1 = 2^{\aleph_0}$
- $|\mathbb{R}| = \aleph_1 = 2^{\aleph_0}$
- $|\mathbb{R} \times \mathbb{R}| = \aleph_1 = 2^{\aleph_0}$
- $|\mathbb{C}| = \aleph_1 = 2^{\aleph_0}$

Un conjunto es *enumerable* si tiene el mismo cardinal de los naturales. Si un conjunto es finito o enumerable entonces se dice que es *contable*. Cuando dos conjuntos tienen el mismo cardinal se dice que son *equipotentes*.

3.3. Ejercicios

1. Sea Ψ_A el siguiente predicado constructor del conjunto A

$$\Psi_A(x) = \begin{cases} V, & \text{si} \quad x \text{ es un dígito y } x \text{ es un multiplo de 3;} \\ F, & \text{en otro caso.} \end{cases}$$

   ¿Cuál es el conjunto listado por extensión que define $\Psi_A(x)$?.

2. Si $A = \{2, 4, 6, 8, 0\}$, definir un predicado constructor Ψ_A para el conjunto A.

3. Encuentre todos los conjuntos A, tales que $A \subseteq B$, para el conjunto $B = \{\bullet, \blacksquare, \blacktriangle, \blacklozenge\}$.

4. Sea $U = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 0, \bullet, \blacksquare, \blacktriangle, \blacklozenge\}$ un universo para los conjuntos $A = \{2, 4, 6, 8, 0, \bullet, \blacktriangle\}$, $B = \{3, 6, 9, 0, \bullet, \blacklozenge\}$ y $C = \{1, 3, 5, 7, 9, \blacksquare, \blacklozenge\}$. Calcule:

   i. $A \cup B$
   ii. $A \cap C$
   iii. $(A \cap B) \cup C$
   iv. $(A \cup C) \cap (B \cup C)$
   v. $\overline{B \cup C}$
   vi. $A \cap \overline{C}$
   vii. $\overline{B} \cap \overline{C}$
   viii. $\overline{A}^B$
   ix. $\overline{A}$
   x. $\overline{A}^{B \cap C}$
   xi. $\overline{A \cap B}^C$
   xii. $\overline{A}^{\overline{B}^{\overline{C}}}$

5. Sean $A = \varnothing$, $B = \{\varnothing\}$, $C = \{\bullet, \blacksquare, \blacktriangle, \blacklozenge\}$ y $D = \big\{\{x\}, \{y\}, \{z\}\big\}$, calcular

   i. $\wp(A)$ ii. $\wp(B)$ iii. $\wp(C)$ iv. $\wp(D)$

6. Sean $A = \{2, 4, 6, 8, 0\}$, $B = \{\bullet, \blacksquare, \blacktriangle, \blacklozenge\}$ y $C = \{\mathrm{a}, \mathrm{e}, \mathrm{i}, \mathrm{o}, \mathrm{u}\}$, calcular:

   i. $A \times B$
   ii. $B \times A$
   iii. $A \times C$
   iv. $C \times A$
   v. $B \times C$
   vi. $C \times B$
   vii. $A \times B \times C$
   viii. $A \times C \times B$
   ix. $B \times A \times C$
   x. $B \times C \times A$
   xi. $C \times A \times B$
   xii. $C \times B \times A$

7. Halle el cardinal de los conjuntos obtenidos en el numeral 5. Para un conjunto A, ¿cuál será el cardinal de $\wp(A)$ en términos de $|A|$?.

8. Halle el cardinal de los conjuntos obtenidos en el numeral 6. Para los conjuntos A, B, C, A_1, A_2, $\ldots$, A_n.

   (a) ¿Cuál será el cardinal de $A \times B$ en términos de $|A|$ y $|B|$?.
   (b) ¿Cuál será el cardinal de $A \times B \times C$ en términos de $|A|$, $|B|$ y $|C|$?.

(c) ¿Cuál será el cardinal de $A_1 \times A_2 \times \cdots \times A_n$ en términos de $|A_1|$, $|A_2|$, ..., $|A_n|$?.

9. Dados dos conjuntos A y B, se tiene que

$$|A \cup B| = |A| + |B| - |A \cap B|,$$

esto por que al sumar $|A|$ y $|B|$ se incluye dos veces $|A \cap B|$, y por lo tanto hay que restarlo. Para el caso de tres conjuntos A, B y C, ¿Cuál es el cardinal del conjunto $A \cup B \cup C$?. *Ayuda*: Utilice diagramas de Venn para tres conjuntos.

10. Sea $\Psi_{A \smallsetminus B}$ el siguiente predicado constructor del conjunto *diferencia* $A \smallsetminus B$

$$\Psi_{A \smallsetminus B}(x) = \begin{cases} V, & (x \in A) \wedge (x \notin B); \\ F, & \text{en otro caso.} \end{cases}$$

(a) Haga un diagrama de Venn que represente la operación $A \smallsetminus B$ entre conjuntos.

(b) Para los conjuntos A, B y C del numeral 4 calcule:

i. $A \smallsetminus B$ iii. $B \smallsetminus A$ v. $C \smallsetminus A$
ii. $A \smallsetminus C$ iv. $B \smallsetminus C$ vi. $C \smallsetminus B$

(c) ¿Cuál es el cardinal del conjunto $A \smallsetminus B$?.

(d) ¿El conjunto $A \smallsetminus B$ es igual a $B \smallsetminus A$?.

11. Sea $\Psi_{A \triangle B}$ el siguiente predicado constructor del conjunto *diferencia simétrica* $A \triangle B$

$$\Psi_{A \triangle B}(x) = \begin{cases} V, & (x \in A) \otimes (x \in B); \\ F, & \text{en otro caso.} \end{cases}$$

(a) Haga un diagrama de Venn que represente la operación $A \triangle B$ entre conjuntos.

(b) Para los conjuntos A, B y C del numeral 4 calcule:

i. $A \triangle B$ iii. $B \triangle A$ v. $C \triangle A$
ii. $A \triangle C$ iv. $B \triangle C$ vi. $C \triangle B$

(c) ¿Cuál es el cardinal del conjunto $A \triangle B$?.

(d) ¿El conjunto $A \triangle B$ es igual a $B \triangle A$?.

Capítulo 4

Introducción a los lenguajes de programación

4.1. Identificadores y variables

Un *identificador* es una secuencia de símbolos que se utilizan como nombres de variables, funciones, clases y otras estructuras de los lenguajes de programación.

Los identificadores se escriben como secuencias de caracteres alfanuméricos del alfabeto inglés o el guión bajo (*underscore*) (_), tales que su primer símbolo no es un dígito. Un identificar valido debe cumplir con la condición adicional de que no pertenezca a las palabras reservadas para el lenguaje, a continuación se listan las palabras reservadas del lenguaje C++:

```
auto     bool     break     case     catch     char     class     const
continue     default      delete      do     double      else      enum
extern     false     float     for     friend     goto     if     inline
int     long     namespace     new     operator      private   protected
public       register       return       short      signed       sizeof
static     struct     switch      template     this      throw      try
true     typedef     union     unsigned     void     volatile     while
```

Las siguientes secuencias de carácteres son ejemplos de identificadores validos:

```
i
x
suma
sumando1
sumando2
Edad
paisDeNacimiento
_nombre
area_circulo
```

Las siguientes secuencias de carácteres son ejemplos de secuencias que no son identificadores, ¿por qué?:

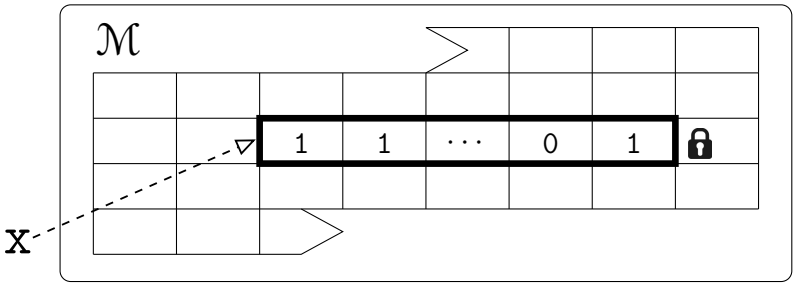
```
1er_mes
primer nombre
while
p@dre
día
```

Una nota importante es que el lenguaje C++ es sensible a mayúsculas y minúsculas, esto quiere decir que por ejemplo los identificadores

```
dia                Dia                DIA
```

sirven para declarar entidades (variable, funciones, etc.) que son diferentes, pues al ser la misma palabra, difiere en que algunas letras son mayúsculas en unos identificadores y en los otros no.

Una *variable* es un espacio de la memoria $\mathcal{M}$ donde se almacena un dato, es un espacio donde se guarda la información necesaria para realizar las acciones que ejecutan los programas.



Para declarar una variable se necesitan principalmente dos componentes, el tipo de variable y el nombre. Los tipos de variables se estudian en la siguiente sección, con respecto al nombre, este simplemente debe ser un identificador valido que no sea una palabra reservada.

En general una variable se declara así

```
T x;
```

Donde $\mathbb{T}$ es el tipo de dato o conjunto al que pertenece la variable y `x` es el identificador que es el nombre de la variable. El símbolo `;` es obligatorio (¡es necesario¡).

Una buena práctica de programación es asignarle el nombre a una variable de tal manera que indique por un lado el papel que desempeña dicha variable en el algoritmo y por otro los posibles valores que almacena. Nombres de variables recomendados dependiendo del tipo de problema pueden ser:

```
velocidad    espacio     masa    aceleracion    exponente    termino1
valor_maximo       area_circulo      nombre_estudiante      last_name
```

4.2. Tipos de datos primitivos

En programación existe un tipo de dato que permite aproximar el conjunto de los números enteros conocido como `int`. Otro tipo de dato que aproxima el conjunto de los números reales se conoce como `double`. Otro tipo de dato conocido como `char` es el que sirve para representar las diferentes letras. Para la representación de los valores de verdad se tendrán los booleanos representados en el tipo de dato `bool`.

Estos tipos de datos son conocidos como primitivos pues están definidos en el lenguaje de programación C++ y porque de ellos se pueden derivar otros tipos de datos definidos por el programador.

4.2.1. Enteros

Los enteros en C++ se codifican con la palabra `int` y su declaración es la siguiente

Si x es una variable algebraica que varia en el conjunto $\mathbb{Z}$, para definir x en el lenguaje C++ se utiliza la expresión

```
int x;
```

lo que sirve para declarar que la variable `x` pertenece a los enteros que son representables en el lenguaje C++.

El subconjunto de los números enteros que pueden ser representados en el lenguaje C++, es el conjunto de enteros con signo que se representan con 32 bits (4 bytes) y que usan un tipo de codificación interna llamada *complemento a 2*, los valores de este conjunto varían en el rango

$$-2147483648 \leq x \leq 2147483647$$

Los literales enteros, es decir, la sintaxis de los valores que pueden ser asignados a las variables de tipo `int` que soporta C++ son por ejemplo:

```
-32768      -0      -1      -127
 32768       0       1       127
+32768      +0      +1      +127
```

Cuando se declara una variable de tipo entero, no se sabe que valor tiene, por eso es necesario inicializar la variable. Los siguientes son ejemplos de inicializaciones de variables de tipo `int`

```
int i = 0;
int j = 1;
int n = 5;
int p = -10;
int k = -1;
```

4.2.2. Reales

Los reales en C++ se codifican con la palabra `double` y su declaración es la siguiente

Si x es una variable algebraica que varia en el conjunto $\mathbb{R}$, para definir x en el lenguaje C++ se utiliza la expresión

```
double x;
```

lo que sirve para declarar que la variable `x` pertenece a los reales que son representables en el lenguaje C++.

El subconjunto de los números reales que pueden ser representados en el lenguaje C++, es un subconjunto propio de los racionales, que se representan con 64 bits (8 bytes) y que usan un tipo de codificación definida por el *IEEE standard for Binary Floating-Point Arithmetic 754* de 1985, los valores distintos de 0 de este conjunto varían en el rango

$$-1.7976931348623157 \times 10^{+308} \leq x \leq -2.2250738585072014 \times 10^{-308}$$

y

$$2.2250738585072014 \times 10^{-308} \leq x \leq 1.7976931348623157 \times 10^{+308}$$

que dan una precisión científica de 15 dígitos.

Los literales reales, es decir, la sintaxis de los valores que pueden ser asignados a las variables de tipo `double` que soporta C++ son por ejemplo:

```
   -3.14159265      -0.0      -6.02214129E+23      -6.674287e-11
    3.14159265       0.0       6.02214129E23       6.674287E-11
   +3.14159265      +0.0      +6.02214129e+23      +6.674287E-11
```

Cuando se declara una variable de tipo real, no se sabe que valor tiene, por eso es necesario inicializar la variable. Los siguientes son ejemplos de inicializaciones de variables de tipo `double`

```
double e = 2.7182818284;
double a = +1.0;
double X = -1.0;
double Luz = 2.998e+8;
double const0 = 1.3806488E-23;
double coordenada_1 = -2.5;
```

4.2.3. Booleanos

Los booleanos en C++ se codifican con la palabra `bool` y su declaración es la siguiente

Si x es una variable algebraica que varia en el conjunto $\mathbb{B}$, para definir x en el lenguaje C++ se utiliza la expresión

```
bool x;
```

lo que sirve para declarar que la variable `x` pertenece al conjunto de los booleanos $\big(\mathbb{B} = \{V, F\}\big)$.

Como sólo hay dos valores de verdad V y F, en C++ sólo hay dos literales para representar los valores lógicos, estos son:

```
true                    false
```

donde la cadena `true` representa el valor de verdad V y la cadena `false` representa el valor de verdad F.

Cuando se declara una variable de tipo booleano, no se sabe que valor tiene, por eso es necesario inicializar la variable. Los siguientes son ejemplos de inicializaciones de variables de tipo `bool`

```
bool b = true;
bool flag = true;
bool exp = false;
bool isPrime = false;
```

4.2.4. Caracteres

Los caracteres representan los símbolos definidos por el ASCII (*American Standard Code for Information Interchange*). Los caracteres se representan con 8 bits (1 byte), lo que ofrece 256 símbolos distintos. El conjunto $ASCII$ cumple con la siguiente característica

$$ASCII \supset \big\{ \texttt{!}, \texttt{"}, \texttt{\#}, \texttt{\$}, \texttt{\%}, \texttt{\&}, \texttt{'}, \texttt{(}, \texttt{)}, \texttt{*}, \texttt{+}, \texttt{,}, \texttt{-}, \texttt{.}, \texttt{/},$$
$$\texttt{0}, \texttt{1}, \texttt{2}, \texttt{3}, \texttt{4}, \texttt{5}, \texttt{6}, \texttt{7}, \texttt{8}, \texttt{9},$$
$$\texttt{:}, \texttt{;}, \texttt{<}, \texttt{=}, \texttt{>}, \texttt{?}, \texttt{@},$$
$$\texttt{A}, \texttt{B}, \texttt{C}, \texttt{D}, \texttt{E}, \texttt{F}, \texttt{G}, \texttt{H}, \texttt{I}, \texttt{J}, \texttt{K}, \texttt{L}, \texttt{M}, \texttt{N}, \texttt{O}, \texttt{P}, \texttt{Q}, \texttt{R}, \texttt{S}, \texttt{T}, \texttt{U}, \texttt{V}, \texttt{W}, \texttt{X}, \texttt{Y}, \texttt{Z},$$
$$\texttt{[}, \texttt{\textbackslash}, \texttt{]}, \texttt{\^{}}, \texttt{\_}, \texttt{`},$$
$$\texttt{a}, \texttt{b}, \texttt{c}, \texttt{d}, \texttt{e}, \texttt{f}, \texttt{g}, \texttt{h}, \texttt{i}, \texttt{j}, \texttt{k}, \texttt{l}, \texttt{m}, \texttt{n}, \texttt{o}, \texttt{p}, \texttt{q}, \texttt{r}, \texttt{s}, \texttt{t}, \texttt{u}, \texttt{v}, \texttt{w}, \texttt{x}, \texttt{y}, \texttt{z},$$
$$\texttt{\{}, \texttt{|}, \texttt{\}}, \texttt{\~{}} \big\}$$

A continuación se presenta una serie de tablas con todos símbolos ASCII.

DEC	Símbolo	Descripción
0	NUL	Null char
1	SOH	Start of Heading
2	STX	Start of Text
3	ETX	End of Text
4	EOT	End of Transmission
5	ENQ	Enquiry
6	ACK	Acknowledgment
7	BEL	Bell
8	BS	Back Space
9	HT	Horizontal Tab
10	LF	Line Feed
11	VT	Vertical Tab
12	FF	Form Feed
13	CR	Carriage Return
14	SO	Shift Out/X-On
15	SI	Shift In/X-Off

DEC	Símbolo	Descripción
16	DLE	Data Line Escape
17	DC1	Device Control 1 (oft. XON)
18	DC2	Device Control 2
19	DC3	Device Control 3 (oft. XOFF)
20	DC4	Device Control 4
21	NAK	Negative Acknowledgement
22	SYN	Synchronous Idle
23	ETB	End of Transmit Block
24	CAN	Cancel
25	EM	End of Medium
26	SUB	Substitute
27	ESC	Escape
28	FS	File Separator
29	GS	Group Separator
30	RS	Record Separator
31	US	Unit Separator

DEC	Símbolo	Descripción
32	[SPACE]	Space
33	!	Exclamation mark
34	"	Double quotes (or speech marks)
35	#	Number
36	$	Dollar
37	%	Procenttecken
38	&	Ampersand
39	’	Single quote
40	(	Open parenthesis (or open bracket)
41	)	Close parenthesis (or close bracket)
42	*	Asterisk
43	+	Plus
44	,	Comma
45	-	Hyphen
46	.	Period, dot or full stop
47	/	Slash or divide

DEC	Símbolo	Descripción
48	0	Zero
49	1	One
50	2	Two
51	3	Three
52	4	Four
53	5	Five
54	6	Six
55	7	Seven
56	8	Eight
57	9	Nine
58	:	Colon
59	;	Semicolon
60	<	Less than (or open angled bracket)
61	=	Equals
62	>	Greater than (or close angled bracket)
63	?	Question mark

DEC	Símbolo	Descripción
64	@	At symbol
65	A	Uppercase A
66	B	Uppercase B
67	C	Uppercase C
68	D	Uppercase D
69	E	Uppercase E
70	F	Uppercase F
71	G	Uppercase G
72	H	Uppercase H
73	I	Uppercase I
74	J	Uppercase J
75	K	Uppercase K
76	L	Uppercase L
77	M	Uppercase M
78	N	Uppercase N
79	O	Uppercase O

DEC	Símbolo	Descripción
80	P	Uppercase P
81	Q	Uppercase Q
82	R	Uppercase R
83	S	Uppercase S
84	T	Uppercase T
85	U	Uppercase U
86	V	Uppercase V
87	W	Uppercase W
88	X	Uppercase X
89	Y	Uppercase Y
90	Z	Uppercase Z
91	[	Opening bracket
92	\	Backslash
93	]	Closing bracket
94	^	Caret-circumflex
95	_	Underscore

DEC	Símbolo	Descripción
96	‘	Grave accent
97	a	Lowercase a
98	b	Lowercase b
99	c	Lowercase c
100	d	Lowercase d
101	e	Lowercase e
102	f	Lowercase f
103	g	Lowercase g
104	h	Lowercase h
105	i	Lowercase i
106	j	Lowercase j
107	k	Lowercase k
108	l	Lowercase l
109	m	Lowercase m
110	n	Lowercase n
111	o	Lowercase o

DEC	Símbolo	Descripción
112	p	Lowercase p
113	q	Lowercase q
114	r	Lowercase r
115	s	Lowercase s
116	t	Lowercase t
117	u	Lowercase u
118	v	Lowercase v
119	w	Lowercase w
120	x	Lowercase x
121	y	Lowercase y
122	z	Lowercase z
123	{	Opening brace
124	|	Vertical bar
125	}	Closing brace
126	~	Equivalency sign - tilde
127	*[DEL]*	Delete

DEC	Símbolo	Descripción
128	€	Euro sign
129		
130	‚	Single low-9 quotation mark
131	ƒ	Latin small letter f with hook
132	„	Double low-9 quotation mark
133	…	Horizontal ellipsis
134	†	Dagger
135	‡	Double dagger
136	ˆ	Modifier letter circumflex accent
137	‰	Per mille sign
138	Š	Latin capital letter S with caron
139	‹	Single left-pointing angle quotation
140	Œ	Latin capital ligature OE
141		
142	Ž	Latin captial letter Z with caron
143		

DEC	Símbolo	Descripción
144		
145	‘	Left single quotation mark
146	’	Right single quotation mark
147	“	Left double quotation mark
148	”	Right double quotation mark
149	•	Bullet
150	–	En dash
151	—	Em dash
152	˜	Small tilde
153	™	Trade mark sign
154	š	Latin small letter S with caron
155	›	Single right-pointing angle quotation mark
156	œ	Latin small ligature oe
157		
158	ž	Latin small letter z with caron
159	Ÿ	Latin capital letter Y with diaeresis

DEC	Símbolo	Descripción
160		Non-breaking space
161	¡	Inverted exclamation mark
162	¢	Cent sign
163	£	Pound sign
164	¤	Currency sign
165	¥	Yen sign
166	¦	Pipe, Broken vertical bar
167	§	Section sign
168	¨	Spacing diaeresis - umlaut
169	©	Copyright sign
170	ª	Feminine ordinal indicator
171	«	Left double angle quotes
172	¬	Not sign
173	-	Soft hyphen
174	®	Registered trade mark sign
175	¯	Spacing macron - overline

DEC	Símbolo	Descripción
176	°	Degree sign
177	±	Plus-or-minus sign
178	²	Superscript two - squared
179	³	Superscript three - cubed
180	´	Acute accent - spacing acute
181	µ	Micro sign
182	¶	Pilcrow sign - paragraph sign
183	·	Middle dot - Georgian comma
184	¸	Spacing cedilla
185	¹	Superscript one
186	º	Masculine ordinal indicator
187	»	Right double angle quotes
188	¼	Fraction one quarter
189	½	Fraction one half
190	¾	Fraction three quarters
191	¿	Inverted question mark

DEC	Símbolo	Descripción
192	À	Latin capital letter A with grave
193	Á	Latin capital letter A with acute
194	Â	Latin capital letter A with circumflex
195	Ã	Latin capital letter A with tilde
196	Ä	Latin capital letter A with diaeresis
197	Å	Latin capital letter A with ring above
198	Æ	Latin capital letter AE
199	Ç	Latin capital letter C with cedilla
200	È	Latin capital letter E with grave
201	É	Latin capital letter E with acute
202	Ê	Latin capital letter E with circumflex
203	Ë	Latin capital letter E with diaeresis
204	Ì	Latin capital letter I with grave
205	Í	Latin capital letter I with acute
206	Î	Latin capital letter I with circumflex
207	Ï	Latin capital letter I with diaeresis

DEC	Símbolo	Descripción
208	Ð	Latin capital letter ETH
209	Ñ	Latin capital letter N with tilde
210	Ò	Latin capital letter O with grave
211	Ó	Latin capital letter O with acute
212	Ô	Latin capital letter O with circumflex
213	Õ	Latin capital letter O with tilde
214	Ö	Latin capital letter O with diaeresis
215	×	Multiplication sign
216	Ø	Latin capital letter O with slash
217	Ù	Latin capital letter U with grave
218	Ú	Latin capital letter U with acute
219	Û	Latin capital letter U with circumflex
220	Ü	Latin capital letter U with diaeresis
221	Ý	Latin capital letter Y with acute
222	Þ	Latin capital letter THORN
223	ß	Latin small letter sharp s - ess-zed

DEC	Símbolo	Descripción
224	à	Latin small letter a with grave
225	á	Latin small letter a with acute
226	â	Latin small letter a with circumflex
227	ã	Latin small letter a with tilde
228	ä	Latin small letter a with diaeresis
229	å	Latin small letter a with ring above
230	æ	Latin small letter ae
231	ç	Latin small letter c with cedilla
232	è	Latin small letter e with grave
233	é	Latin small letter e with acute
234	ê	Latin small letter e with circumflex
235	ë	Latin small letter e with diaeresis
236	ì	Latin small letter i with grave
237	í	Latin small letter i with acute
238	î	Latin small letter i with circumflex
239	ï	Latin small letter i with diaeresis

DEC	Símbolo	Descripción
240	ð	Latin small letter eth
241	ñ	Latin small letter n with tilde
242	ò	Latin small letter o with grave
243	ó	Latin small letter o with acute
244	ô	Latin small letter o with circumflex
245	õ	Latin small letter o with tilde
246	ö	Latin small letter o with diaeresis
247	÷	Division sign
248	ø	Latin small letter o with slash
249	ù	Latin small letter u with grave
250	ú	Latin small letter u with acute
251	û	Latin small letter u with circumflex
252	ü	Latin small letter u with diaeresis
253	ý	Latin small letter y with acute
254	þ	Latin small letter thorn
255	ÿ	Latin small letter y with diaeresis

Los *ASCII* en C++ se codifican con la palabra `char` y su declaración es la siguiente

Si x es una variable algebraica que varia en el conjunto *ASCII*, para definir x en el lenguaje C++ se utiliza la expresión

```
char x;
```

lo que sirve para declarar que la variable `x` pertenece al conjunto de los *ASCII*.

Existen algunos caracteres especiales que no tiene su propio símbolo en el teclado o que no se imprime el símbolo en la pantalla o que tienen un uso particular en C++ (son reservados) y que son utilizados comúnmente; estos caracteres se representan de la siguiente manera, usando el símbolo \ (*back slash*) como símbolo auxiliar:

`\n` : Nueva línea.

`\t` : Tabulador horizontal.

`\\` : Diagonal invertida (back slash).

`\'` : Imprime apóstrofo.

`\"` : Imprime Comillas.

`\b` : Retroceso (retrocede un espacio el cursor).

`\v` : Tabulador vertical (coloca el cursor justo debajo del último carácter de la línea actual).

`\r` : Retorno de carro (coloca el cursor en el primer carácter de la línea actual y sobreescribe el texto de la línea).

`\?` : Imprime el símbolo de interrogación.

Cuando se declara una variable de tipo carácter, no se sabe que valor tiene, por eso es necesario inicializar la variable. Los siguientes son ejemplos de inicializaciones de variables de tipo `char`, para indicar que se está definiendo un literal de carácter, se encierra el símbolo entre
apóstrofos `' '`, así como se muestra a continuación:

```
char c = ' ';
char CH = '\n';
char letra = 'a';
char caracter = 'A';
char value = '\"';
char _last = '\'';
char C_0 = '&';
char cero = '0';
```

4.3. Operadores y expresiones aritméticas

4.3.1. Operadores aritméticos

Para los datos de tipo numérico se pueden utilizar los siguientes operadores infijos, a excepción del operador `-` que puede actuar también como un operador prefijo:

`+` : Suma de dos valores, por ejemplo, cuando se evalúa la expresión `2.0 + 3.0` se obtiene el valor 5.0.

`-` : Resta de dos valores, por ejemplo, cuando se evalúa la expresión `2.0 - 3.0` se obtiene el valor -1.0. También se utiliza para cambiar el signo de un número si se utiliza con un sólo operando, por ejemplo, cuando se evalúa la expresión `-23` se obtiene el valor -23.

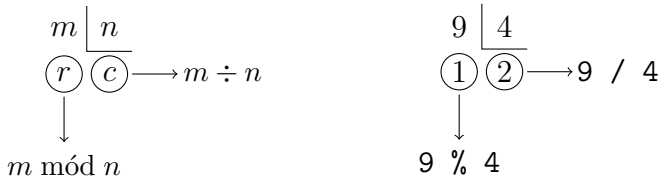
`*` : Multiplicación de dos valores, por ejemplo, cuando se evalúa la expresión `2.0 * -3.0` se obtiene el valor -6.0.

`/` : División de dos valores, cuando alguno de los operandos es real retorna la división exacta, por ejemplo, cuando se evalúa la expresión `-3.0/2` se obtiene el valor -1.5. Cuando ambos operandos son enteros, se obtiene la parte entera de la división exacta, por ejemplo, cuando se evalúa la expresión `-3/2` se obtiene el valor -1. El valor del segundo operando debe ser distinto de 0.

`%` : El resto de la división de dos números, representa la operación matemática

$$m \text{ mód } n = r,$$

por ejemplo, cuando se evalúa la expresión `9 % 4` se obtiene el valor 1, que es lo mismo que 9 mód 4 = 1, el residuo de dividir 9 entre 4.



Otro ejemplo es `3 % 4 = 3`, el residuo de dividir 3 entre 4 es 3.

4.3.2. Operadores de asignación

Para asignar valores a variables se pueden utilizar los siguientes operadores infijos:

`=` : Asignación. La parte de la izquierda que debe ser una variable. Sirve para almacenar un dato en una variable. Asigna el valor de evaluar la parte de la derecha a la variable de la parte de la izquierda. Por ejemplo, cuando se evalúa la expresión `pi = 3.14159265`, entonces se almacena el valor 3.14159265 en la variable `pi`.

`+=` : Asignación con suma. La parte de la izquierda debe ser una variable. Suma la evaluación de parte de la derecha con el valor almacenado en la variable definida en la parte de la izquierda y guarda el resultado en la variable de parte de la izquierda. Por ejemplo, la expresión `x += 2`, es equivalente a la expresión `x = x + 2`.

`-=` : Asignación con resta. La parte de la izquierda debe ser una variable. Resta al valor almacenado en la variable definida en la parte de la izquierda el resultado de la evaluación de parte de la derecha y guarda el resultado en la variable de parte de la izquierda. Por ejemplo, la expresión `x -= 2`, es equivalente a la expresión `x = x - 2`.

`*=` : Asignación con multiplicación. La parte de la izquierda debe ser una variable. Multiplica el valor almacenado en la variable definida en la parte de la izquierda con la evaluación de parte de la derecha y guarda el producto en la variable de parte de la izquierda. Por ejemplo, la expresión `x *= 2`, es equivalente a la expresión `x = x * 2`.

`/=` : Asignación con división. La parte de la izquierda debe ser una variable. Divide el valor almacenado en la variable definida en la parte de la izquierda entre el valor de la evaluación de la parte de la derecha y guarda el resultado en la variable de parte de la izquierda. Por ejemplo, la expresión `x /= 2`, es equivalente a la expresión `x = x / 2`. El valor de la evaluación de la parte de la derecha debe ser distinto de 0.

`%=` : Asignación con residuo. La parte de la izquierda debe ser una variable. Calcula el residuo de dividir el valor almacenado en la variable definida en la parte de la izquierda entre el valor de la evaluación de la parte de la derecha y guarda el resultado en la variable de parte de la izquierda. Por ejemplo, la expresión `x %= 2`, es equivalente a la expresión `x = x % 2`. El valor de la evaluación de la parte de la derecha debe ser distinto de 0.

Dos de los operadores más utilizados para asignar valores a variables en programación son los siguientes operadores posfijos:

`++` : Incremento en una unidad y asignación, se utiliza como un operador unario y actúa sólo sobre variables, por ejemplo, si la variable `i` almacena el valor 0, cuando se evalúa la expresión `i++` el valor que almacenará la variable ahora será igual a 1. Es equivalente a la expresión `i = i + 1` o a la expresión `i += 1`.

`--` : Decremento en una unidad y asignación, se utiliza como un operador unario y actúa sólo sobre variables, por ejemplo, si la variable `i` almacena el valor 2, cuando se evalúa la expresión `i--` el valor que almacenará la variable ahora será igual a 1. Es equivalente a la expresión `i = i - 1` o a la expresión `i -= 1`.

4.3.3. Operadores lógicos

`!` : Operador $\neg$ de la negación.

$$!\alpha \Leftrightarrow \neg\alpha$$

`&&` : Operador $\wedge$ de la conjunción.

$$\alpha \texttt{ \&\& } \beta \Leftrightarrow \alpha \wedge \beta$$

`||` : Operador $\vee$ de la disyunción.

$$\alpha \texttt{ || } \beta \Leftrightarrow \alpha \vee \beta$$

4.3.4. Operadores de igualdad y relacionales

`==` : Devuelve V si dos valores son iguales.

$$\alpha \texttt{ == } \beta \Leftrightarrow \alpha = \beta$$

`!=` : Devuelve V si dos valores son distintos.

$$\alpha \texttt{ != } \beta \Leftrightarrow \alpha \neq \beta$$

`>` : Mayor que, devuelve V si el primer operador es estrictamente mayor que el segundo.

$$\alpha \texttt{ > } \beta \Leftrightarrow \alpha > \beta$$

`<` : Menor que, devuelve V si el primer operador es estrictamente menor que el segundo.

$$\alpha \texttt{ < } \beta \Leftrightarrow \alpha < \beta$$

`>=` : Mayor o igual, devuelve V si el primer operador es mayor o igual que el segundo.

$$\alpha \texttt{ >= } \beta \Leftrightarrow \alpha \geq \beta$$

`<=` : Menor igual, devuelve V si el primer operador es menor o igual que el segundo.

$$\alpha \texttt{ <= } \beta \Leftrightarrow \alpha \leq \beta$$

Ejemplo. Para escribir la expresión $x \in (0, 1]$ en el lenguaje C++ se utiliza la sentencia

```
(0 < x && x <= 1)
```

4.3.5. Precedencia de operadores

En la tabla 4.1 se presenta la prioridad de los principales operadores de C++, la prioridad más alta es la 1 y la más baja es la 10.

Operador(es)	Prioridad
`()`	1
`++` `--`	2
`!` `-`*(signo menos)* `+`*(signo más)*	3
`*` `/` `%`	4
`+` `-`	5
`<` `>` `<=` `>=`	6
`==` `!=`	7
`&&`	8
`||`	9
`=` `+=` `-=` `*=` `/=` `%=`	10

Tabla 4.1. Precedencia de los operadores en C++.

Ejemplo. Hallar el valor de la siguiente expresión teniendo en cuenta la prioridad de operadores y que los operandos son números reales

```
12.0 * 3.0 + 4.0 - 8.0 / 2.0 % 3.0
```

i) $(12.0 * 3.0) + 4.0 - 8.0$ / 2.0 % 3.0 ($*$ prioridad 4)

ii) $(12.0 * 3.0) + 4.0 - (8.0$ / $2.0)$ % 3.0 (/ prioridad 4)

iii) $(12.0 * 3.0) + 4.0 - ((8.0 \ / \ 2.0) \ \% \ 3.0)$ (% prioridad 4)

iv) $((12.0 * 3.0) + 4.0) - ((8.0 \ / \ 2.0) \ \% \ 3.0)$ (+ prioridad 5)

v) $(((12.0 * 3.0) + 4.0) - ((8.0 \ / \ 2.0) \ \% \ 3.0))$ (− prioridad 5)

$$\begin{aligned} 12.0 * 3.0 + 4.0 - 8.0 \ / \ 2.0 \ \% \ 3.0 &= 36.0 + 4.0 - 8.0 \ / \ 2.0 \ \% \ 3.0 \\ &= 36.0 + 4.0 - 4.0 \ \% \ 3.0 \\ &= 36.0 + 4.0 - 1.0 \\ &= 40.0 - 1.0 \\ &= 39.0 \end{aligned}$$

Ejemplo. Hallar el valor de la siguiente expresión teniendo en cuenta la prioridad de operadores y que los operandos son números enteros

```
(-2 + 5 % 3 * 4) / 4 + 2
```

i) $(-2 + (5 \ \% \ 3) * 4) \ / \ 4 + 2$ (% prioridad 4)

ii) $(-2 + ((5 \ \% \ 3) * 4)) \ / \ 4 + 2$ (∗ prioridad 4)

iii) $(-2 + ((5 \ \% \ 3) * 4)) \ / \ 4 + 2$ (+ prioridad 5)

iv) $((-2 + ((5 \ \% \ 3) * 4)) \ / \ 4) + 2$ (/ prioridad 4)

v) $(((-2 + ((5 \ \% \ 3) * 4)) \ / \ 4) + 2)$ (+ prioridad 5)

$$\begin{aligned} (-2 + 5 \ \% \ 3 * 4) \ / \ 4 + 2 &= (-2 + 2 * 4) \ / \ 4 + 2 \\ &= (-2 + 8) \ / \ 4 + 2 \\ &= 6 \ / \ 4 + 2 \\ &= 1 + 2 \\ &= 3 \end{aligned}$$

4.4. Evaluación de secuencias de expresiones

Como se vio previamente las cadenas `=`, `+=`, `-=`, `*=`, `/= %=` sirven para representar los operadores de asignación, es decir, permiten asignar un valor a una determinada variable, donde la variable se encuentra a la izquierda del operador y el valor resulta de evaluar una expresión que se encuentra a la derecha del operador. Se debe tener cuidado que al evaluar una expresión el resultado sea del mismo tipo de la variable a la que se le esta asignando el valor, esto es, que a una variable de tipo `int` se le asigne un valor entero, que a una variable de tipo `double` se le asigne un valor real, que a una variable de tipo `bool` se le asigne un valor booleano, que a una variable de tipo `char` se le asigne un carácter, etc.

Una asignación comprende dos partes: el valor al que se le asignar el valor y la expresión. Se espera que el resultado de la evaluación de la expresión sea del mismo tipo de la

expresión. En este sentido se podría extender a expresiones de tipo lógico, aritmético ó carácter.

El proceso de asignar valores a variables es el objetivo central a la hora de construir un programa, ya que un programa no es más que una función que transforma la memoria desde un estado inicial hasta un estado final donde se encuentra el resultado que se quería calcular.

Visto así, en programación, la asignación es una operación temporal, que primero lee el valor de las variables existentes en la memoria, a partir de estos valores se evalúa la expresión a la derecha de la asignación y luego se realiza la asignación a la variable de la izquierda correspondiente al resultado de la evaluación de la expresión, es decir, se actualiza el valor de la variable en la memoria.

Ejemplo. Supónga que un programa contiene las variables `x`, `y` y `z` en el instante de tiempo t, que sus valores en este instante de tiempo son:

$$\texttt{x} = 3, \qquad \texttt{y} = 4 \qquad \text{y} \qquad \texttt{z} = -2$$

si a partir de estos valores se realiza la asignación

```
x = y + z - 2;
```

entonces se tendría que en el instante de tiempo t se evalúa la expresión `y + z - 2` y el resultado de esta evaluación se asignaría a la variable `x` pero ya en el instante de tiempo $t + 1$. Con lo que los valores de las variables (memoria) en este nuevo instante de tiempo seria:

$$\texttt{x} = 0, \qquad \texttt{y} = 4 \qquad \text{y} \qquad \texttt{z} = -2$$

Ejemplo. Supónga que se desea realizar la asignación

```
x = x + 3 - 2 * y
```

cuando $\texttt{x} = 3$ y $\texttt{y} = 5$.

Para entender como se realiza la asignación es útil subindizar las variables teniendo en cuenta el instante de tiempo en el cual se esta leyendo o modificando la memoria. Con base en lo anterior, la expresión se reescribe de la siguiente manera

$$\texttt{x}_{t+1} = \texttt{x}_t + 3 - 2 * \texttt{y}_t$$

Así, si en el instante de tiempo t se tiene que $\texttt{x} = 3$ y $\texttt{y} = 5$, entonces en el instante de tiempo $t + 1$, se tendrá que $\texttt{x} = -4$ y $\texttt{y} = 5$.

Cuando se desea estudiar el comportamiento de una secuencia de asignaciones es útil utilizar una tabla de la *traza* de ejecuciones.

En esta tabla se tiene una columna donde se ubica el instante de tiempo t que inicialmente debe ser igual a 0, en las otras columnas se ubican todas las variables que intervienen en los cálculos. Para las variables que tienen un valor inicial, este valor se ubica en la misma fila del instante de tiempo $t = 0$ y para el resto de variables se utiliza el símbolo — para indicar que aún está indefinido.

Tras iniciar la ejecución de las instrucciones, se va actualizando el valor de las variables a las que se les haya hecho alguna asignación, teniendo en cuenta el instante de tiempo en el cual se realiza la asignación. Esto se realiza hasta que se ejecute toda la secuencia de instrucciones.

Ejemplo. La siguiente tabla es la traza obtenida tras ejecutar la instrucción

```
x = x + 3 - 2 * y
```

cuando `x` $= 3$ y `y` $= 5$.

t	`x`	`y`
0	3	5
1	-4	5

Ejemplo. Supónga que se desea ejecutar la siguiente secuencia de instrucciones

```
i = k + 1;
j = 2 * k;
i = i * k * j;
j = j * k - i;
```

cuando `k` $= 1$. Entonces la traza de la ejecución será la siguiente

t	`k`	`i`	`j`
0	1	—	—
1	1	2	—
2	1	2	2
3	1	4	2
4	1	4	-2

4.5. Ejercicios

1. Construya la traza para la siguiente secuencia de instrucciones:

```
int m = 1;
int n = 2;
int k;
k = m * n;
n = n + 1;
k = m * n;
n = n + 1;
k = m * n;
n = n + 1;
k = m * n;
n = n + 1;
k = m * n;
n = n + 1;
```

Capítulo 5

Relaciones y funciones

5.1. Relaciones

Una *relación* R de A en B es un subconjunto del producto cartesiano $A \times B$, es decir, R se dice relación si $R \subseteq A \times B$. Al conjunto A se le denomina *conjunto de salida* y al conjunto B se le denomina *conjunto de llegada*.

Ejemplo. Sean $A = \{0, 1, 2\}$ y $B = \{\clubsuit, \blacklozenge, \heartsuit, \spadesuit\}$, una de la posibles relaciones que se pueden definir es $R = \big\{(0, \clubsuit), (0, \spadesuit), (2, \spadesuit), (2, \blacklozenge)\big\}$, el cual es un subconjunto de parejas ordenadas pertenecientes al producto cartesiano entre A y B.

El conjunto vacío $\varnothing$ puede considerarse una relación al ser subconjunto de cualquier conjunto. De forma similar, el producto cartesiano $A \times B$ también es una relación pues todo conjunto es subconjunto de si mismo ($A \times B \subseteq A \times B$).

Dada una relación $R \subseteq A \times B$, el conjunto

$$Dom_R = \big\{a : (a \in A) \wedge (\exists b \in B) \wedge \big((a, b) \in R\big)\big\}$$

se denomina el *dominio* de la relación R, a los elementos de Dom_R se les denomina *preimágenes* de la relación R.

Ejemplo. Sean $A = \{0, 1, 2\}$ y $B = \{\clubsuit, \blacklozenge, \heartsuit, \spadesuit\}$, dos conjuntos y sea $R = \big\{(0, \clubsuit), (0, \spadesuit), (2, \spadesuit), (2, \blacklozenge)\big\}$, una relación definida de A en B, el conjunto $Dom_R = \{0, 2\}$ es el dominio de la relación y los elementos 0 y 2 son las preimágenes de la relación R.

Dada una relación $R \subseteq A \times B$, el conjunto

$$Ran_R = \big\{b : (b \in B) \wedge (\exists a \in A) \wedge \big((a, b) \in R\big)\big\}$$

se denomina el *rango* o *codominio* de la relación R, a los elementos de Ran_R se les denomina *imágenes* de la relación R.

Ejemplo. Sean $A = \{0, 1, 2\}$ y $B = \{\clubsuit, \blacklozenge, \heartsuit, \spadesuit\}$ dos conjuntos y sea $R = \big\{(0, \clubsuit), (0, \spadesuit), (2, \spadesuit), (2, \blacklozenge)\big\}$ una relación definida de A en B, el conjunto $Ran_R = \{\clubsuit, \blacklozenge, \spadesuit\}$ es el rango de la relación y los elementos $\clubsuit$, $\blacklozenge$ y $\spadesuit$ son las imágenes de la relación R.

Dentro de una relación es posible vincular un elemento de un conjunto con más de un elemento de otro conjunto, no tener elementos en la relación o incluir sólo algunos elementos que pertenecen al producto cartesiano de los dos conjuntos. En el ejemplo anterior, no se tienen elementos que incluyan el elemento 1 del conjunto de salida A ni el elemento ♥ del conjunto de llegada B.

Si se tiene una relación R de un conjunto cartesiano $R \subseteq A \times B$, ésta puede notarse como $R : A \rightsquigarrow B$ esto evita la notación de subconjunto pero aclara que se esta realizando una asignación de elementos de A en elementos de B.

Una de las características importantes del concepto de relación es su noción de agrupar elementos de diferentes tipos. Esta forma de conectar elementos de diferentes conjuntos tiene mucha aplicación en programación, pues dicha relación puede ser vista como una asignación.

La representación gráfica de una relación se puede hacer en términos de *diagramas Sagitales*. En la figura 5.1 se muestra una representación de la relación $R = \{(0, \clubsuit), (0, \spadesuit), (2, \spadesuit), (2, \diamondsuit)\}$ definida en el ejemplo anterior. Como se puede apreciar cada flecha representa un vinculo entre cada uno de los elementos del conjunto A y los del conjunto B.

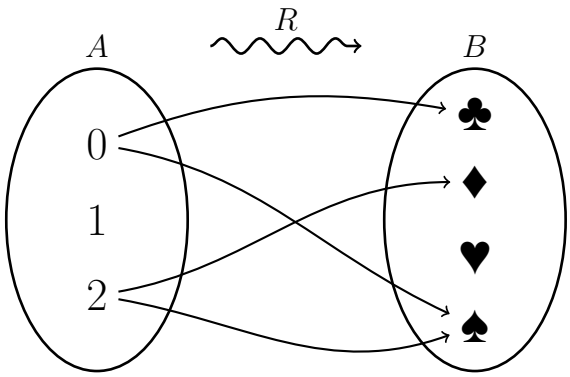


FIGURA 5.1. Representación de la relación $R = \{(0, \clubsuit), (0, \spadesuit), (2, \spadesuit), (2, \diamondsuit)\}$ mediante diagramas Sagitales.

Hay relaciones que se pueden establecer entre elementos del mismo conjunto, es decir, una relación que cumpla que $R \subseteq A \times A$

Ejemplo. Si se tiene el conjunto $A = \{\clubsuit, \diamondsuit, \heartsuit, \spadesuit\}$, se puede construir la relación $R = \{(\clubsuit, \diamondsuit), (\diamondsuit, \clubsuit), (\heartsuit, \spadesuit), (\spadesuit, \heartsuit)\}$. En la figura 5.2 se muestra una representación de esta relación.

5.1.1. Propiedades de las relaciones

Una de las características más importantes e interesantes que poseen las relaciones que están definidas sobre el mismo conjunto, son las propiedades que pueden cumplir; las cuales permiten definir los conceptos de relaciones de orden y de equivalencia entre los elementos de un conjunto. Dichas propiedades son las siguientes:

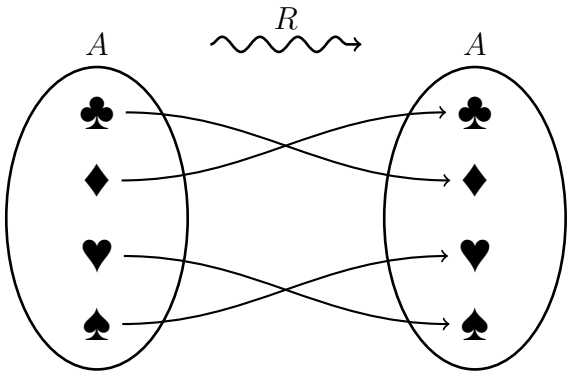


FIGURA 5.2. Representación de la relación $R = \{(\clubsuit, \blacklozenge), (\blacklozenge, \clubsuit), (\heartsuit, \spadesuit), (\spadesuit, \heartsuit)\}$ mediante diagramas Sagitales.

Reflexiva: se dice que una relación $R \subseteq A \times A$ es *reflexiva*, si y sólo si,

$$(\forall a \in A)\big((a, a) \in R\big).$$

Ejemplo. Dada la relación $R = \{(0,0), (0,1), (1,1), (1,2), (2,2)\}$ construida sobre el mismo conjunto $A = \{0, 1, 2\}$ se puede afirmar que

- La relación R es reflexiva pues están todas las parejas de los elementos vinculados consigo mismos $(0, 0)$, $(1, 1)$ y $(2, 2)$.

Simetrica: se dice que una relación es *simétrica*, si y sólo si,

$$\text{Si} \quad (a, b) \in R \quad \text{entonces} \quad (b, a) \in R.$$

Ejemplo. Dada la relación $R = \{(0,0), (0,1), (1,1), (1,2), (2,2)\}$ construida sobre el mismo conjunto $A = \{0, 1, 2\}$ se puede afirmar que

- La relación R no es simétrica pues está la pareja $(0, 1)$ pero no está la pareja $(1, 0)$. En este caso es suficiente con un caso que no se de para decir que una relación no cumple la propiedad.

Antisimetrica: se dice que una relación es *antisimétrica*, si y sólo si,

$$\text{Si} \quad \big((a, b) \in R\big) \wedge \big((b, a) \in R\big) \quad \text{entonces} \quad a = b.$$

Esta definición es equivalente a que

$$\text{Si} \quad \big((a, b) \in R\big) \wedge (a \neq b) \quad \text{entonces} \quad (b, a) \notin R.$$

Ejemplo. Dada la relación $R = \{(0,0), (0,1), (1,1), (1,2), (2,2)\}$ construida sobre el mismo conjunto $A = \{0, 1, 2\}$ se puede afirmar que

- La relación R es antisimétrica pues cumple con la definición, ya que las tres parejas $(0, 0)$, $(1, 1)$ y $(2, 2)$ son de la forma (a, a) que cumplen con la definición, y para las otras parejas $(0, 1)$ y $(1, 2)$ no están presentes las parejas $(1, 0)$ o $(2, 1)$, por lo que también se cumplen con la definición de antisimetría.

Transitiva: una relación se dice *transitiva*, si y sólo si,

$$\text{Si} \quad \big((a,b) \in R\big) \wedge \big((b,c) \in R\big) \quad \text{entonces} \quad (a,c) \in R.$$

Esta noción puede relacionarse con el silogismo hipotético en el cual si $(\alpha \to \beta) \wedge (\beta \to \gamma)$, entonces se puede concluir que $\alpha \to \gamma$.

Ejemplo. Dada la relación $R = \big\{(0,0),(0,1),(1,1),(1,2),(2,2)\big\}$ construida sobre el mismo conjunto $A = \{0,1,2\}$ se puede afirmar que

- La relación R no es transitiva pues están las parejas $(0,1)$ y $(1,2)$, pero no está la pareja $(0,2)$.

El hecho de que una relación no sea simétrica no quiere decir que la relación sea antisimétrica y viceversa.

Ejemplo. Si se tiene el conjunto $A = \{0,1,2\}$, y sobre éste se define la relación $R = \big\{(0,0),(0,1),(1,0),(1,2),(2,2)\big\}$, se tiene que la relación no es simétrica pues la pareja $(1,2)$ está, pero la pareja $(2,1)$ no está, ni es antisimétrica pues las parejas $(0,1)$ y $(1,0)$ están, y es sabido que $1 \neq 0$.

Ejemplo. Si se tiene el conjunto $A = \{0,1,2\}$, y sobre éste se define la relación $R = \big\{(0,0),(1,1),(2,2)\big\}$, se tiene que la relación es tanto simétrica como antisimétrica. ¿Por qué?.

5.1.2. Relaciones de orden

Una relación R definida sobre el mismo conjunto, se dice que es una *relación de orden*[1], si y sólo si, es reflexiva, antisimétrica y transitiva. Una relación de orden se suele notar con el símbolo $\leq_A$.

Ejemplo. Si se tiene el conjunto $A = \{0,1,2\}$, y la relación $R = \big\{(0,1),(0,2),(0,0),(1,1),(1,2),(2,2)\big\}$ esta es una relación de orden, pues es reflexiva, antisimétrica y transitiva.

Una relación antisimétrica y transitiva es un *orden estricto*. Un orden estricto se nota como $<_A$.

Un *preorden* es una relación que es reflexiva y transitiva.

Una *relación de orden total*, es una relación de orden para la cual para todo $a, b \in A$ se tiene que $a \leq_A b$ ó $b \leq_A a$.

5.1.3. Relaciones de equivalencia

Una relación R se dice que es una *relación de equivalencia*, si y sólo si, es reflexiva, simétrica y transitiva. El símbolo utilizado para decir que una relación es equivalente es $\equiv_A$.

Dado el conjunto $A = \{0,1,2\}$ y una relación R definida sobre A de la siguiente manera $R = \big\{(0,0),(0,1),(1,0),(1,1),(2,2)\big\}$, se puede observar que R es una relación de equivalencia, pues:

[1]También se suele denominar *relación de orden parcial*.

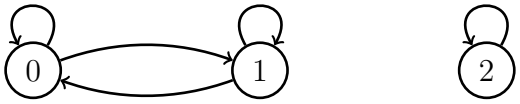
Es reflexiva: las parejas $(0,0)$, $(1,1)$ y $(2,2)$ están.

Es simétrica: además de las parejas $(0,0)$, $(1,1)$ y $(2,2)$, como la pareja $(0,1)$ está en la relación, entonces la pareja $(1,0)$ debería estar, como efectivamente ocurre, el caso reciproco es similar.

Es transitiva: como las parejas $(0,1)$ y $(1,0)$ están, debería estar la pareja $(0,0)$, y de forma análoga, como están las parejas $(1,0)$ y $(0,1)$, debería estar la pareja $(1,1)$; para el resto de los casos, como las parejas que quedan son aquellas que tienen igual la primera y la segunda componente, estas se relacionan consigo mismas, por lo que la transitividad se obtiene de forma directa.

Las relaciones de equivalencia tienen una particular característica, es que los elementos que están relacionados se pueden interpretar como que representan el mismo objeto. Así, una relación de equivalencia define una *partición* del conjunto en grupos de conjuntos con propiedades similares; tales como que no son vacíos, que no tienen elementos en común, y que la unión de todos los grupos es el conjunto inicial. A estos grupos se les conoce como *clases de equivalencia*.

Ejemplo. Para el conjunto $A = \{0,1,2\}$ y la relación de equivalencia definida sobre el mismo conjunto $R = \big\{(0,0),(0,1),(1,0),(1,1),(2,2)\big\}$, aquí se observa que se pueden formar los grupos $\{0,1\}$ y $\{2\}$, los cuales son una partición del conjunto A y que definen dos clases de equivalencia. Obsérvese que los valores 0 y 1, resultan ser equivalentes, es decir, que representan el mismo objeto.



5.2. Función parcial

Definición. Una relación $f : A \rightsquigarrow B$ se dice *función parcial* si y sólo si,

$$\text{Si} \quad \big((x,y) \in f\big) \wedge \big((x,y') \in f\big) \quad \text{entonces} \quad y = y'.$$

El concepto de función parcial es tan fundamental en matemáticas (computación) que tiene su propia notación, en vez de notarlas como $f : A \rightsquigarrow B$, las funciones parciales se notan así $f : A \to B$. Esta notación representa que a un elemento del conjunto de salida A le corresponde uno y sólo un elemento en el conjunto de llegada B. Cuando a un elemento $x \in A$ le corresponde un elemento $y \in B$, a través de la función f se suele usar la notación

$$f(x) = y$$

para expresar que $(x,y) \in f$. Cuando se desea especificar tanto el conjunto de salida como el conjunto de llegada se usa la *notación dominio–rango*

$$\begin{aligned} f : A &\to B \\ x &\mapsto f(x) \end{aligned}$$

Ejemplo. Para los conjuntos $A = \{0, 1, 2, 3, 4\}$ y $B = \{\clubsuit, \blacklozenge, \heartsuit, \spadesuit\}$. En la figura 5.3 se representan las parejas $\big\{(0, \blacklozenge), (1, \blacklozenge), (4, \heartsuit)\big\}$ que definen una relación. Esta relación seria una función, pues si a un elemento del conjunto de salida A le corresponde un elemento del conjunto B, éste es único. 0 y 1 están asociados con $\blacklozenge$ y 4 con el símbolo $\heartsuit$. Nótese que no importa que 0 y 1 estén asociados al mismo elemento de B, que pueden haber elementos en A que no se estén asociados a elementos de B como ocurre con 2 y 3, y que pueden haber y elementos de B que no se estén asociados a elementos de A, tal como ocurre con $\clubsuit$ y $\spadesuit$.

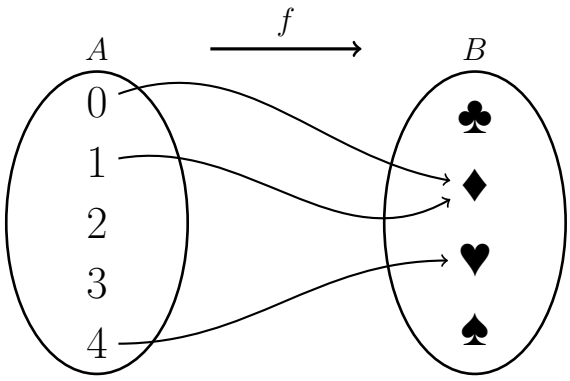


FIGURA 5.3. Representación de la función $f = \big\{(0, \blacklozenge), (1, \blacklozenge), (4, \heartsuit)\big\}$ mediante diagramas Sagitales.

Ejemplo. Si a la función $f = \big\{(0, \clubsuit), (1, \clubsuit), (2, \heartsuit)\big\}$ se adicionara la pareja $(1, \spadesuit)$, esta nueva relación $f' = \big\{(0, \clubsuit), (1, \clubsuit), (2, \heartsuit), (1, \spadesuit)\big\}$ dejaría de ser función, pues a 1 le corresponderían dos valores diferentes $\clubsuit$ y $\spadesuit$. En la figura 5.4 se representa la relación f'.

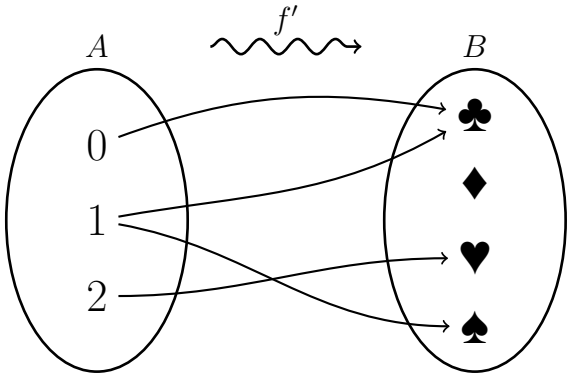


FIGURA 5.4. Representación de la relación $f' = \big\{(0, \clubsuit), (1, \clubsuit), (2, \heartsuit), (1, \spadesuit)\big\}$ mediante diagramas Sagitales.

5.2.1. Propiedades de las funciones

Las funciones también poseen propiedades muy interesantes, así como las de las relaciones de un conjunto en si mismo, las más importantes de estas son:

Inyectiva: una función $f : A \to B$ se dice *inyectiva* o *uno a uno* si y sólo si,

$$\text{Si} \quad (x, z) \in f \wedge (y, z) \in f \quad \text{entonces} \quad x = y.$$

A un elemento del conjunto B le corresponde una sola preimagen del conjunto A.

Ejemplo. En la figura 5.5 se muestra una representación de la función inyectiva $f = \big\{(0, \spadesuit), (2, \clubsuit)\big\}$.

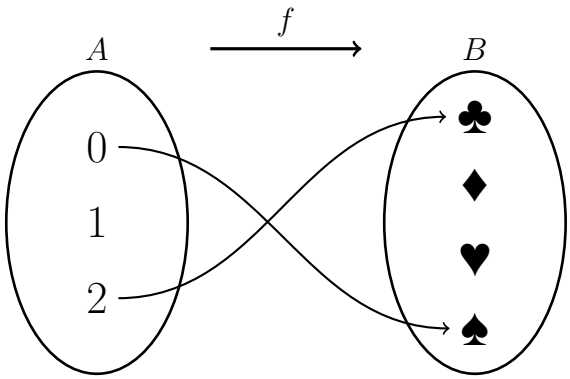


FIGURA 5.5. Representación de la función inyectiva $f = \big\{(0, \spadesuit), (2, \clubsuit)\big\}$ mediante diagramas Sagitales.

Sobreyectiva: una función $f : A \to B$ se dice *sobreyectiva* o *suprayectiva* si y sólo si,

$$(\forall b \in B)(\exists a \in A) \quad \text{tal que} \quad \big((a, b) \in f\big).$$

Cada elemento del conjunto B tiene una preimagen del conjunto A.

Ejemplo. En la figura 5.6 se muestra una representación de la función sobreyectiva $f = \big\{(0, \clubsuit), (1, \heartsuit), (2, \spadesuit), (4, \clubsuit)\big\}$.

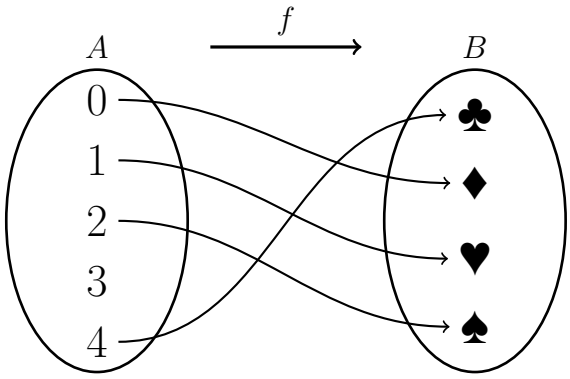


FIGURA 5.6. Representación de la función sobreyectiva $f = \big\{(0, \clubsuit), (1, \heartsuit), (2, \spadesuit), (4, \clubsuit)\big\}$ mediante diagramas Sagitales.

Total: una función $f : A \to B$ se dice *total* si y sólo si,

$$(\forall a \in A)(\exists b \in B) \quad \text{tal que} \quad \big((a,b) \in f\big).$$

Cada elemento del conjunto A tiene una imagen del conjunto B.

Ejemplo. En la figura 5.7 se muestra una representación de la función total $f = \big\{(0,\spadesuit),(1,\diamondsuit),(2,\clubsuit),(3,\spadesuit),(4,\diamondsuit)\big\}$.

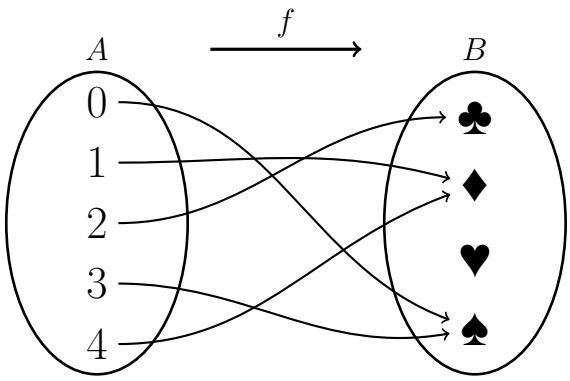


FIGURA 5.7. Representación de la función total $f = \big\{(0,\spadesuit),(1,\diamondsuit),(2,\clubsuit),(3,\spadesuit),(4,\clubsuit)\big\}$ mediante diagramas Sagitales.

Biyectiva: una función $f : A \to B$ se dice *biyectiva* si y sólo si, f es inyectiva, sobreyectiva y total.

Ejemplo. En la figura 5.8 se muestra una representación de la función biyectiva $f = \big\{(0,\spadesuit),(1,\diamondsuit),(2,\spadesuit),(3,\clubsuit)\big\}$.

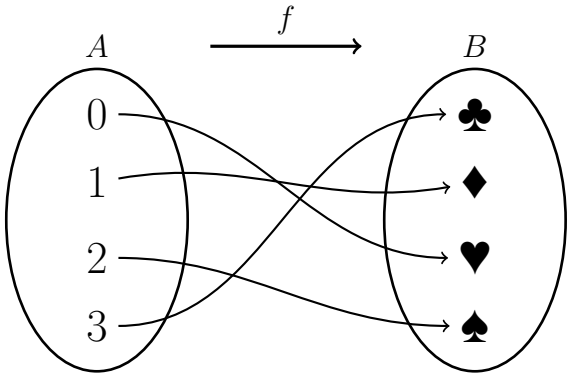
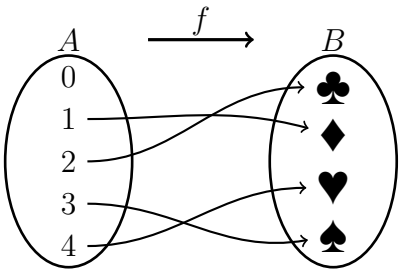


FIGURA 5.8. Representación de la función biyectiva $f = \big\{(0,\spadesuit),(1,\diamondsuit),(2,\spadesuit),(3,\clubsuit)\big\}$ mediante diagramas Sagitales.

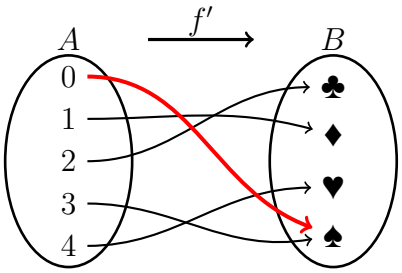
5.3. Extensión de una función parcial a una función total

Las funciones parciales suelen presentarse en computación cuando en el problema estudiado se presenta un evento excepcional con algunos valores para los cuales la función se encuentra indefinida y por lo tanto no se puede evaluar.

Todas las funciones parciales se pueden extender de tal manera que estas sean un subconjunto de una función total. Para obtener la función total se tienen varias alternativas, las cuales se explicarán con base en la siguiente función parcial.



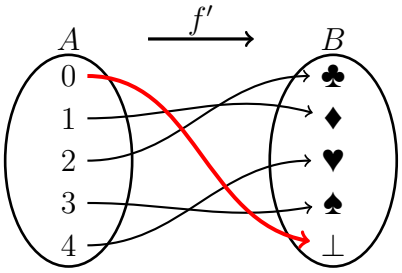
Una alternativa para obtener una función total a partir de una función parcial es la de asignar los valores del conjunto de salida que no pertenecen al dominio a algún valor del conjunto de llegada, como se muestra a continuación



Asignar un valor al conjunto de salida no siempre es posible o conveniente debido a que pueden introducirse valores incoherentes o errados con respecto al problema estudiado.

Una alternativa es adicionar al conjunto de llegada un valor nuevo que indique un error dado un valor en el conjunto de salida. En la función total al obtenerse este nuevo valor, se quiere notar que en el problema estudiado se ha encontrado un caso de error o un problema excepcional.

Típicamente el valor con el que se extiende el conjunto de llegada es el símbolo $\perp$, el cual representa una contradicción, en estos casos un error o un caso excepcional.



Ejemplo. Para la siguiente función div de $\mathbb{R} \times \mathbb{R}$ en $\mathbb{R}$,

$$\begin{aligned} \text{div} : \mathbb{R} \times \mathbb{R} &\to \mathbb{R} \\ (x, y) &\mapsto \frac{x}{y}, \end{aligned}$$

se tiene que si la segunda proyección de la pareja del dominio (x, y) es igual a 0, entonces la función se encuentra indefinida, por lo tanto la función no es total.

Para hallar una función que contenga la anterior y que sea total, se adicional el símbolo $\perp$ al conjunto de llegada y se extiende la función de la siguiente manera

$$\begin{aligned} \text{div} : \mathbb{R} \times \mathbb{R} &\to \mathbb{R} \cup \{\perp\} \\ (x, y) &\mapsto \begin{cases} \perp, & \text{si } y = 0; \\ \frac{x}{y}, & \text{en otro caso.} \end{cases} \end{aligned}$$

5.4. Funciones importantes en computación

Definición. Identidad
La función identidad id_A relaciona a cada elemento de un conjunto A consigo mismo, de la siguiente manera

$$\begin{aligned} id_A : A &\to A \\ x &\mapsto id_A(x) = x \end{aligned}$$

Ejemplo. En la figura se muestra una representación de la función identidad $id_A = \{(\clubsuit, \clubsuit), (\blacklozenge, \blacklozenge), (\heartsuit, \heartsuit), (\spadesuit, \spadesuit)\}$ para el conjunto $A = \{\clubsuit, \blacklozenge, \heartsuit, \spadesuit\}$.

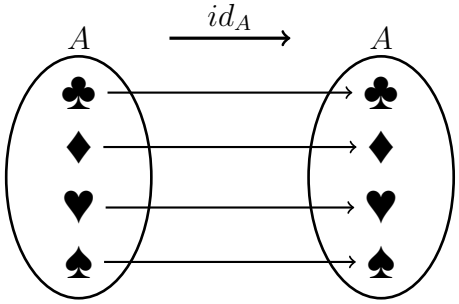


FIGURA 5.9. Representación de la función identidad $id_A = \{(\clubsuit, \clubsuit), (\blacklozenge, \blacklozenge), (\heartsuit, \heartsuit), (\spadesuit, \spadesuit)\}$ mediante diagramas Sagitales.

Definición. Valor absoluto
La función *valor absoluto* de x se denota como $|x|$ y se define como:

$$\begin{aligned} |x| : \mathbb{R} &\to \mathbb{R}^{0,+} \\ x &\mapsto \begin{cases} x, & \text{si} \quad x \geq 0; \\ -x, & \text{en otro caso.} \end{cases} \end{aligned}$$

Ejemplos. $|-3.14| = 3.14$, $|3.14| = 3.14$, $|-1| = 1$, $|1| = 1$, $|0| = 0$, $|\pi| = \pi$, $|-\pi| = \pi$, $\left|-\sqrt{2}\right| = \sqrt{2}$, $\left|\sqrt{2}\right| = \sqrt{2}$.

Definición. Potencia
La función *potencia* de b elevado al exponente n se denota como b^n y se define como:

$$b^n : \mathbb{R} \times \mathbb{Z} \to \mathbb{R} \cup \{\perp\}$$

$$(b,n) \mapsto \begin{cases} 1, & \text{si} \quad (b \neq 0) \wedge (n = 0); \\ \underbrace{b * b * \cdots * b}_{n\text{–veces}}, & \text{si} \quad (b \neq 0) \wedge (n \in \mathbb{Z}^+); \\ \dfrac{1}{b^{|n|}}, & \text{si} \quad (b \neq 0) \wedge (n \in \mathbb{Z}^-); \\ 0, & \text{si} \quad (b = 0) \wedge (n > 0); \\ \perp, & \text{si} \quad (b = 0) \wedge (n \leq 0). \end{cases}$$

Ejemplos. $1^0 = 1$, $2^0 = 1$, $2^3 = 8$, $3^{-2} = \frac{1}{9}$, $0.5^3 = 0.125$, $\left(\frac{2}{3}\right)^2 = \frac{4}{9}$, $0^0 = \perp$, $0^{-1} = \perp$.

Definición. Raíz cuadrada
La función *raíz cuadrada* de x se denota como $\sqrt{x}$ y se define como:

$$\sqrt{x} : \mathbb{R}^{0,+} \to \mathbb{R}^{0,+}$$
$$x \mapsto b, \quad \text{donde} \quad b^2 = x.$$

Ejemplos. $\sqrt{0} = 0$, $\sqrt{1} = 1$, $\sqrt{4} = 2$, $\sqrt{9} = 3$, $\sqrt{25} = 5$, $\sqrt{0.25} = 0.5$, $\sqrt{\frac{4}{9}} = \frac{2}{3}$,

$$\sqrt{2} \approx 1.41421356237309504880168872420969807856967187537694807317667973799 \cdots .$$

En programación se define que $x^{1/2} = \sqrt{x}$ y se puede demostrar que $\sqrt{x^2} = |x|$.

Definición. Logaritmo
La función *logaritmo* en base b de x se denota como $\log_b x$ y se define como:

$$\log_b x : \mathbb{R}^+ \times \mathbb{R}^+ \to \mathbb{R}$$
$$(b,x) \mapsto y, \quad \text{donde} \quad b^y = x.$$

Ejemplos. $\log_2 4 = 2$, $\log_2 8 = 3$, $\log_2 1 = 0$ (en general $\log_b 1 = 0$), $\log_e e = 1$ (en general $\log_b b = 1$), $\log_3 \frac{1}{9} = -2$, $\log_{0.5} 0.125 = 3$, $\log_{0.25} 2 = -\frac{1}{2}$.

Definición. Piso
La función *piso* de x se denota como $\lfloor x \rfloor$ y se define como:

$$\lfloor x \rfloor : \mathbb{R} \to \mathbb{Z}$$
$$x \mapsto n, \quad \text{donde} \quad (n \in \mathbb{Z}) \wedge (n \leq x < n + 1).$$

es decir, $\lfloor x \rfloor$ es el mayor entero que es menor o igual a x.

Ejemplos. $\lfloor -3.141516 \rfloor = -4$, $\lfloor -3 \rfloor = -3$, $\lfloor -1.5 \rfloor = -2$, $\lfloor -0.1 \rfloor = -1$, $\lfloor 0 \rfloor = 0$, $\lfloor 0.1 \rfloor = 0$, $\lfloor 1.5 \rfloor = 1$, $\lfloor 3 \rfloor = 3$, $\lfloor 3.141516 \rfloor = 3$, $\lfloor 1.\overline{3} \rfloor = 1$.

Definición. Techo
La función *techo* de x se denota como $\lceil x \rceil$ y se define como:

$$\lceil x \rceil : \mathbb{R} \to \mathbb{Z}$$
$$x \mapsto n, \quad \text{donde} \quad (n \in \mathbb{Z}) \wedge (n - 1 < x \leq n).$$

es decir, $\lceil x \rceil$ es el menor entero que es mayor o igual a x.

Ejemplos. $\lceil -3.141516 \rceil = -3$, $\lceil -3 \rceil = -3$, $\lceil -1.5 \rceil = -1$, $\lceil -0.1 \rceil = 0$, $\lceil 0 \rceil = 0$, $\lceil 0.1 \rceil = 1$, $\lceil 1.5 \rceil = 2$, $\lceil 3 \rceil = 3$, $\lceil 3.141516 \rceil = 4$, $\lceil 1.\overline{3} \rceil = 2$.

Definición. Parte entera
La función *parte entera* de x se denota como $[x]$ y se define como:

$$\begin{aligned} [x] : \mathbb{R} &\to \mathbb{Z} \\ x &\mapsto \begin{cases} \lfloor x \rfloor, & \text{si} \quad x \geq 0; \\ \lceil x \rceil, & \text{si} \quad x < 0. \end{cases} \end{aligned}$$

Ejemplos. $[-3.141516] = -3$, $[-3] = -3$, $[-1.5] = -1$, $[-0.1] = 0$, $[0] = 0$, $[0.1] = 0$, $[1.5] = 1$, $[3] = 3$, $[3.141516] = 3$, $\left[1.\overline{3}\right] = 1$.

Definición. Parte fraccionaria
La función *parte fraccionaria* de x se denota como $\text{frac}(x)$ y se define como:

$$\begin{aligned} \text{frac}(x) : \mathbb{R} &\to [0, 1) \\ x &\mapsto |x| - \left\lfloor |x| \right\rfloor. \end{aligned}$$

Ejemplos. $\text{frac}(-3.141516) = 0.141516$, $\text{frac}(-3) = 0.0$, $\text{frac}(-1.5) = 0.5$, $\text{frac}(-0.1) = 0.1$, $\text{frac}(0) = 0.0$, $\text{frac}(0.1) = 0.1$, $\text{frac}(1.5) = 0.5$, $\text{frac}(3) = 0.0$, $\text{frac}(3.141516) = 0.141516$, $\text{frac}(1.\overline{3}) = 0.\overline{3}$.

Definición. Redondeo
La función *redondeo* de x se denota como $\text{round}(x)$, retorna el entero más próximo al número x. Para los reales no negativos retorna el techo si la parte fraccionaria es mayor o igual a 0.5, retorna el piso si la parte fraccionaria es menor a 0.5. Para los reales negativos retorna el piso si la parte fraccionaria es mayor o igual a 0.5, retorna el techo si la parte fraccionaria es menor a 0.5. Formalmente esto se define como:

$$\begin{aligned} \text{round}(x) : \mathbb{R} &\to \mathbb{Z} \\ x &\mapsto \begin{cases} \lceil x \rceil, & \text{si} \quad (x \geq 0) \wedge \big(\text{frac}(x) \geq 0.5\big); \\ \lfloor x \rfloor, & \text{si} \quad (x \geq 0) \wedge \big(\text{frac}(x) < 0.5\big); \\ -\,\text{round}(-x), & \text{si} \quad x < 0. \end{cases} \end{aligned}$$

Ejemplos. $\text{round}(-3.141516) = -3, \text{round}(-3) = -3, \text{round}(-1.5) = -2, \text{round}(-0.1) = 0$, $\text{round}(0) = 0$, $\text{round}(0.1) = 0$, $\text{round}(1.5) = 2$, $\text{round}(3) = 3$, $\text{round}(3.141516) = 3$, $\text{round}(-1.8) = -2$, $\text{round}(1.8) = 2$, $\text{round}(1.\overline{3}) = 1$, $\text{round}(1.\overline{5}) = 2$.

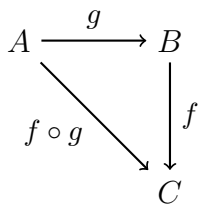
5.5. Composición de funciones

Definición. Sean $g : A \to B$ y $f : B \to C$ dos funciones, si se cumple que $Ran_g \subseteq Dom_f$, entonces es posible definir una nueva función llamada la *composición* de f y g, la cual se denota como $f \circ g$ y que se define como

$$\begin{aligned} f \circ g : A &\to C \\ a &\longmapsto f(g(a)) \end{aligned}$$

Así definida la composición, se tiene que $Dom_{f\circ g} = Dom_g$ y $Ran_{f\circ g} \subseteq Ran_f$.

Una representación que permite entender mejor como opera ésta nueva función, es utilizando un diagrama conmutativo, como se muestra a continuación:



Ejemplo. Sea $A = \{1, 2, 3, 4, 5, 6\}$, $B = \{\clubsuit, \blacklozenge, \heartsuit, \spadesuit, \clubsuit\}$ y $C = \{a, b, c, d\}$ tres conjuntos. Si $g = \big\{(1, \heartsuit), (2, \blacklozenge), (3, \clubsuit), (4, \clubsuit), (6, \clubsuit)\big\}$ y $f = \big\{(\clubsuit, c), (\blacklozenge, d), (\heartsuit, a), (\spadesuit, b), (\clubsuit, c)\big\}$ entonces $f \circ g = \big\{(1, a), (2, d), (3, c), (4, c), (6, c)\big\}$; aquí también se observa que $Dom_{f\circ g} = Dom_g = \{1, 2, 3, 4, 6\}$ y $Ran_{f\circ g} = \{a, c, d\} \subseteq Ran_f = \{a, b, c, d\}$. En las figuras 5.10 y 5.11 se muestra una representación de la composición de las funciones f y g.

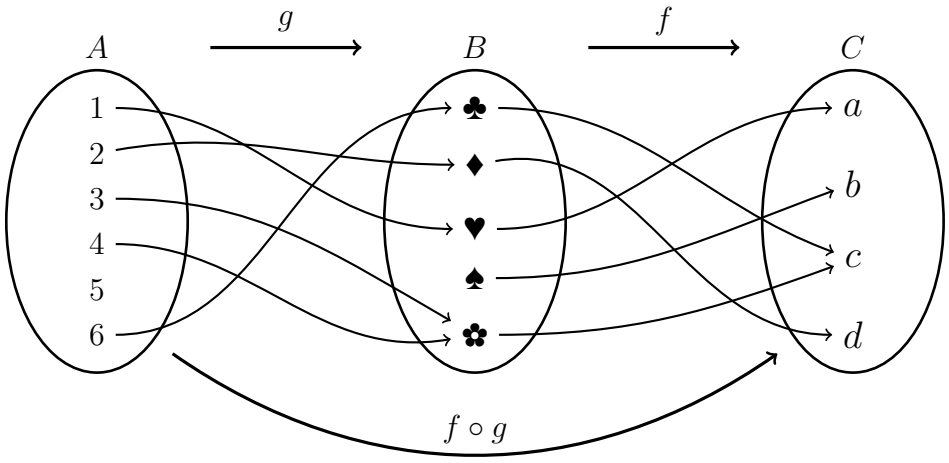


FIGURA 5.10. Representación mediante diagramas Sagitales de la composición de las funciones f y g, $f \circ g = \big\{(1, a), (2, d), (3, c), (4, c), (6, c)\big\}$.

Ejemplo. Sean g y f las siguientes funciones:

$$\begin{aligned} g : \mathbb{R} &\to \mathbb{Z} \\ x &\mapsto \lfloor x \rfloor \end{aligned} \qquad\qquad \begin{aligned} f : \mathbb{Z} &\to \mathbb{Z} \\ n &\mapsto n * n \end{aligned}$$

entonces la función compuesta de las funciones f y g será

$$\begin{aligned} f \circ g : \mathbb{R} &\to \mathbb{Z} \\ x &\mapsto f\big(g(x)\big) = f\big(\lfloor x \rfloor\big) = \lfloor x \rfloor * \lfloor x \rfloor \end{aligned}$$

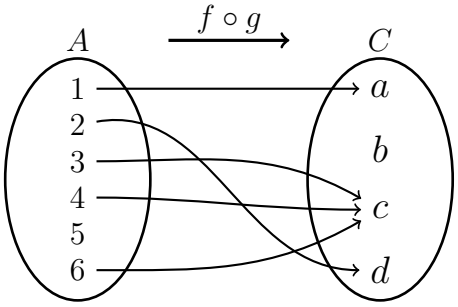


FIGURA 5.11. Representación mediante diagramas de la función $f \circ g = \{(1,a),(2,d),(3,c),(4,c),(6,c)\}$.

5.5.1. Evaluación como composición de funciones

Una secuencia de instrucciones se puede entender matemáticamente como la composición de una serie de funciones, donde cada vez que se haga uso de una variable, ésta se reemplaza por la asignación inmediatamente anterior, sí existe una asignación de esta variable.

Ejemplo. Para la siguiente secuencia de instrucciones

```
i = k + 1;
j = 2 * k;
i = i * k * j;
j = j * k - i;
```

se puede observar que utilizando la funciones $+$, $*$, $-$ como funciones binarias usando notación prefija, se obtienen las siguientes expresiones matemáticas:

$$
\begin{aligned}
i &= +(k,1)\\
j &= *(2,k)\\
i &= *\big(*(i,k),j\big) = *\big(*\big(+(k,1),k\big),*(2,k)\big)\\
j &= -\big(*(j,k),i\big) = -\big(*\big(*(2,k),k\big),*\big(*\big(+(k,1),k\big),*(2,k)\big)\big)
\end{aligned}
$$

Obsérvese que la última expresión está escrita únicamente en términos de k y si se evalúa en el valor $k = 1$, entonces se obtiene el valor para j que igual a -2, el cual es el valor obtenido en el último ejemplo del la sección de evaluación de expresiones.

$$
\begin{aligned}
j &= -\big(* \big(* (2,k), k\big), *\big(* \big(+ (k,1), k\big), *(2,k)\big)\big)\\
j &= -\big(* (*(2,1),1), *\big(* (+(1,1),1), *(2,1)\big)\big)\\
j &= -\big(* (2,1), *\big(* (+(1,1),1), *(2,1)\big)\big)\\
j &= -\big(2, *\big(* (+(1,1),1), *(2,1)\big)\big)\\
j &= -\big(2, *\big(* (2,1), *(2,1)\big)\big)\\
j &= -\big(2, *\big(2, *(2,1)\big)\big)\\
j &= -\big(2, *\big(2,2\big)\big)\\
j &= -\big(2,4\big)\\
j &= -2
\end{aligned}
$$

5.6. Ejercicios

1. Sean $A = \{0, 1, 2\}$ y $B = \{\bullet, \blacksquare, \blacktriangle, \blacklozenge\}$ dos conjuntos. ¿Cuántas relaciones de A en B existen?.

2. Sean $A = \{0, 1\}$ y $B = \{\bullet, \blacktriangle\}$ dos conjuntos. Encuentre todas las relaciones de A en B.

3. Sea $A = \{\bullet, \blacksquare, \blacktriangle, \blacklozenge\}$ un conjunto. ¿Cuántas relaciones de A en A existen?.

4. Sea $A = \{\bullet, \blacksquare, \blacktriangle, \blacklozenge\}$ un conjunto. Represente mediante un diagrama sagital las relaciones que se presentan a continuación.

   i. $R_1 = \big\{(\bullet,\bullet), (\bullet,\blacksquare), (\blacksquare,\bullet), (\blacksquare,\blacksquare), (\blacktriangle,\blacklozenge), (\blacklozenge,\bullet), (\blacklozenge,\blacklozenge)\big\}$

   ii. $R_2 = \big\{(\bullet,\bullet), (\bullet,\blacksquare), (\blacksquare,\bullet)\big\}$

   iii. $R_3 = \big\{(\bullet,\bullet), (\bullet,\blacksquare), (\bullet,\blacklozenge), (\blacksquare,\bullet), (\blacksquare,\blacksquare), (\blacktriangle,\blacktriangle), (\blacklozenge,\bullet), (\blacklozenge,\blacklozenge)\big\}$

   iv. $R_4 = \big\{(\blacksquare,\bullet), (\blacktriangle,\bullet), (\blacktriangle,\blacksquare), (\blacklozenge,\bullet), (\blacklozenge,\blacksquare), (\blacklozenge,\blacktriangle)\big\}$

   v. $R_5 = \big\{(\bullet,\bullet), (\bullet,\blacksquare), (\bullet,\blacktriangle), (\bullet,\blacklozenge), (\blacksquare,\blacksquare), (\blacksquare,\blacktriangle), (\blacksquare,\blacklozenge), (\blacktriangle,\blacktriangle), (\blacktriangle,\blacklozenge), (\blacklozenge,\blacklozenge)\big\}$

   vi. $R_6 = \big\{(\blacktriangle,\blacklozenge)\big\}$

5. De las relaciones del numeral 4. ¿Cuál es el dominio?, ¿Cuál es el rango?.

6. De las relaciones del numeral 4. ¿Cuáles son reflexivas?, ¿Cuáles son simétricas?, ¿Cuáles son antisimétricas?, ¿Cuáles son transitivas?.

7. De las relaciones del numeral 4. ¿Cuáles son una relación de orden?, ¿Cuáles son una relación de equivalencia?.

8. De las siguientes funciones definidas de $\mathbb{Z}$ a $\mathbb{Z}$

   i. $f_1(n) = 1$
   ii. $f_2(n) = n$
   iii. $f_3(n) = n^2$
   iv. $f_4(n) = n^3$
   v. $f_5(n) = 2^n$
   vi. $f_6(n) = \log_2 n$

   ¿Cuál es el dominio?, ¿Cuál es el rango?.

9. De las funciones definidas en el numeral 8. ¿Cuáles son inyectivas?, ¿cuáles son sobreyectivas?, ¿cuáles son totales?, ¿cuáles son biyecciones?.

10. De las siguientes relaciones definidas de $\mathbb{Z}$ en $\mathbb{R}$

    i. $f_1(x) = x$
    ii. $f_2(x) = x^2$
    iii. $f_3(x) = \sqrt{x}$
    iv. $f_4(x) = |x|$
    v. $f_5(x) = \pm x$
    vi. $f_6(x) = 1/x$

vii. $f_7(x) = 1/x^2$ viii. $f_8(x) = \log_2 x$

¿Cuáles son funciones?.

11. De las relaciones definidas en el numeral 10. ¿Cuál es el dominio?, ¿Cuál es el rango?.

12. De las relaciones definidas en el numeral 10 que son funciones. ¿Cuáles son inyectivas?, ¿cuáles son sobreyectivas?, ¿cuáles son totales?, ¿cuáles son biyecciones?.

Capítulo 6

Funciones en programación y la estructura condicional

En programación, así como en matemáticas, para las funciones definidas como $f : A \to B$, al conjunto A se le denomina dominio y al conjunto B como rango. A partir de estos objetos se construye el encabezado de las funciones de programación.

Sobre esta función se tiene que f corresponde al nombre de la función, el conjunto A corresponde al tipo de los argumentos de dicha función y el conjunto B que es el rango corresponderá al valor de retorno de dicha función.

Ejemplo. *Cuadrado de un número*

Se definirá una función que eleve un número al cuadrado. Para expresar una función que calculé esta operación, en primera instancia se construye la expresión $f : \mathbb{R} \to \mathbb{R}$ que define la función tiene como entrada (dominio) un número real y como salida (rango) un número real. La declaración de la función junto con su cuerpo quedará de la siguiente forma

$$\begin{aligned} f &: \mathbb{R} \to \mathbb{R} \\ (x) &\mapsto x^2 \end{aligned}$$

Esta función también podría notarse como

$$f = \big\{(x, x^2) : (x \in \mathbb{R})\big\}.$$

Dicha función se traduce en lenguaje C++ paso a paso de la siguiente forma:

- Primero se escribe el tipo de retorno (rango). Como es real, el tipo de dato es `double`.

```
double
```

- Posteriormente se escribe el nombre de la función `f`.

```
double f
```

- Entre paréntesis se coloca el tipo y la variable de las variables del dominio. En este caso solamente se tiene la variable x quedando `double x` y se coloca una llave abierta `{`.

```
double f(double x){
```

- En las siguientes líneas se escribe la definición de la función entre las llaves terminando con punto y coma `;` cada linea de la función. Dicha definición de función corresponde al algoritmo ó computo, para generar la imagen calculada de la función se utiliza la palabra clave `return`, de la siguiente manera

```
double f(double x){
  return x * x;
```

- Después de escribir la función se cierra la llave `}` seguida de un punto y coma `;`.

```
double f(double x){
  return x * x;
};
```

Otra posible definición de la función podría escribirse almacenando el valor del producto `x * x` en una variable de tipo `double` y retornándolo así:

```
double f(double x){
  double y;
  y = x * x;
  return y;
};
```

Esto es similar al caso cuando se utiliza la notación

$$f(x) = y$$

donde x es una variable independiente y y es la variable dependiente.

Ejemplo. *Área de un círculo*

Para el desarrollo de esta función lo primero es determinar el nombre. La función se llamará *area_circulo* cuyo dominio es el conjunto de los números reales (para el radio) y cuyo rango pertenece al conjunto de los números reales (el valor de retorno que corresponde al área del círculo).

Teniendo en cuenta que el algoritmo para el cálculo del área de un círculo depende del valor de su radio, entonces, el área del círculo está dada por la expresión $A_c = \pi * r^2$, donde las variables están definidas así:

$$r := \text{Radio del círculo}$$
$$A_c := \text{Área del círculo de radio } r$$

entonces, el planteamiento matemático de la función solicitada será el siguiente

$$\begin{aligned} area\_circulo : \mathbb{R} &\to \mathbb{R} \\ (r) &\mapsto 3.14159265 * r * r \end{aligned}$$

Nótese que para definir el valor π se tomó como aproximación el valor 3.14159265. En vez de elevar el valor del radio al cuadrado se tomó la multiplicación de $r * r$ ya que la potencia no es una operación matemática básica. En la sección de recursividad se definirá potencia como una función.

Esta función se traduce al lenguaje C++ paso a paso de la siguiente forma:

- Primero se escribe el tipo de retorno en este caso el conjunto que corresponde al tipo de dato del área del círculo, como es real el tipo de dato es `double`.

```
double
```

- Posteriormente se escribe el nombre de la función `area_circulo`.

```
double area_circulo
```

  obsérvese que la palabra `circulo` en el nombre de la función se escribe sin tilde, ya que únicamente se pueden utilizar carácteres del alfabeto inglés

- Entre paréntesis se coloca el tipo y la variable de las variables del dominio. En este caso solamente se tiene la variable r de tipo `double r` y se coloca una llave abierta `{`.

```
double area_circulo(double r){
```

- En las siguientes líneas se escribe el cálculo del área del círculo y se retorna el valor calculado.

```
double area_circulo(double r){
  return 3.14159265 * r * r;
```

- Después de escribir la función se cierra la llave `}` seguida de un punto y coma `;`.

```
double area_circulo(double r){
  return 3.14159265 * r * r;
};
```

Otra posible definición de la función podría escribirse almacenando el valor del área del círculo en una variable de tipo `double` y retornándolo así:

```
double area_circulo(double r){
  double area;
  area = 3.14159265 * r * r;
  return area;
};
```

Esto es similar al caso cuando se utiliza la notación

$$area\_circulo(r) = area$$

para expresar la función, la variable independiente y la dependiente.

6.1. Compilación y ejecución de funciones

Para el desarrollo de programas en este libro se utilizará como lenguaje de programación el lenguaje C++. En esta primera parte del libro se definirán las funciones en el mismo archivo fuente. Los archivos de C++ corresponden a la extensión `*.cpp`.

Para la generación de archivos ejecutables se utilizará la salida y entrada estándar de la consola definida en la librería `iostream`.

Todo programa ejecutable elaborado en C++ tiene una función principal que realiza los llamados a las distintas funciones tanto del lenguaje como las definidas por el programador. Se utilizará el siguiente esqueleto para definir dichas funciones en C++

```
#include<iostream>
#include<cstdlib>

using namespace std;

/*
  En esta parte se definen las funciones
*/

int main(){
  /*
    En esta parte se realiza la lectura de datos,
    y los llamados a las funciones
  */
  cout << "\n";
  system("pause"); //windows
  return EXIT_SUCCESS;
};
```

Para el caso del área del círculo, el archivo fuente en C++ sería el siguiente, en el cual se han separado el encabezado del archivo, la función del cálculo del área y la función principal

```
#include<iostream>
#include<cstdlib>

using namespace std;
```

```
double area_circulo(double r){
  return 3.1415962 * r * r;
};
```

```
int main(){
  double radio;
  cout << "radio del circulo? = ";
  cin >> radio;
  cout << "El area del circulo es: ";
  cout << area_circulo(radio);
  cout << "\n";
  system("pause"); //windows
  return EXIT_SUCCESS;
};
```

Como se aprecia en el programa anterior, se utilizan las funciones `cin` y `cout` que pertenecen a la librería de entrada y salida de flujos de datos `iostream`. Las dos primeras líneas de código permiten al compilador incluir las definiciones de estas funciones en el programa principal. La función `cout` lo que hace es mostrar una cadena de caracteres (texto) en la consola. La instrucción `cout << "\n"` lo que hace es imprimir un carácter de fin de línea en la consola. La función `cin` lo que hace es asignar el valor que sea digitado en la consola a la variable de la parte de la izquierda del operador `<<`. En la instrucción `cin >> radio;` se asigna a la variable `radio` el valor digitado por el usuario. `system("pause")` lo que hace es ejecutar el comando *pause* de Windows que permite pedirle al usuario que presione una tecla para continuar la ejecución del programa o en este caso por estar antes del retorno o fin de la función principal para salir de él devolviendo al sistema operativo un valor de `EXIT_SUCCESS` para indicar que ejecución del programa fue satisfactoria, es decir, que en este caso el programa se ejecutó correctamente.

El programa anterior pide al usuario digitar el radio del círculo y almacena el valor de la lectura en la variable radio. Posteriormente despliega un mensaje y se realiza el llamado a la función `area_circulo`. Nótese que los nombres de variables (en este caso radio) en el programa principal no necesariamente deben coincidir con los de la declaración de la función (en este caso la función recibe un valor de tipo `double` llamado `r`), sin embargo se debe notar que su tipo de dato sí debe ser el mismo.

6.2. Funciones con más de un parámetro de entrada

Las funciones están definidas de la forma $f : A \to B$, siendo A y B conjuntos. Esta definición nos permite utilizar productos cartesianos generalizados como dominio en la declaración de las funciones.

Ejemplo. *Área de un rectángulo*

La función que calcula el área de un rectángulo es una función que tiene como parámetros de entrada el ancho y el largo del rectángulo que son valores de tipo real y retorna como salida el área del rectángulo. Lo que puede escribirse como

$$area\_rectangulo : \mathbb{R} \times \mathbb{R} \to \mathbb{R}.$$

Para el calculo del área de un rectángulo es necesario conocer el largo y el ancho del rectángulo, a partir de los cuales el área del rectángulo está dada por la expresión $A_r = l*a$, donde las variables están definidas así:

$$\begin{aligned} l &:= \text{Largo del rectángulo} \\ a &:= \text{Ancho del rectángulo} \\ A_r &:= \text{Área del rectángulo de largo } l \text{ y ancho } a \end{aligned}$$

entonces, la función matemática queda definida de la siguiente forma

$$\begin{aligned} area\_rectangulo : \mathbb{R} \times \mathbb{R} &\to \mathbb{R} \\ (l, a) &\mapsto l * a \end{aligned}$$

Para traducir a lenguaje C++ se toman las mismas reglas de traducción:

- Primero se escribe el tipo de retorno en este caso el conjunto que corresponde al tipo de dato del área del rectángulo, como es real el tipo de dato es `double`.

```
double
```

- Posteriormente se escribe el nombre de la función `area_rectangulo`.

```
double area_rectangulo
```

- Entre paréntesis se colocan el conjunto de cada variable del dominio y el nombre de la variable separados por comas. En este se tienen las variables l y a correspondientes al largo y al ancho del rectángulo y se coloca una llave abierta.

```
double area_rectangulo(double l, double a){
```

- En las siguientes líneas se escribe el cálculo del área del rectángulo, se retorna el valor calculado y se cierra la definición de la función.

```
double area_rectangulo(double l, double a){
  return l * a;
};
```

Otra posible escritura de la función puede ser

```
double area_rectangulo(double l, double a){
  double area;
  area = l * a;
  return area;
};
```

Esto es similar al caso cuando se utiliza la notación

$$area\_rectangulo(l, a) = area$$

para expresar la función, las variables independientes y la dependiente.

Para el llamado de la función se podría tener el siguiente programa principal:

```
#include<iostream>
#include<cstdlib>

using namespace std;
```

```
double area_rectangulo(double l, double a){
  return l * a;
};
```

```
int main(){
  double largo;
  double ancho;
  cout << "largo? = ";
  cin >> largo;
  cout << "ancho? = ";
  cin >> ancho;
  cout << "El area del rectangulo es: ";
  cout << area_rectangulo(largo, ancho);
  cout << "\n";
  system("pause");
  return EXIT_SUCCESS;
};
```

6.3. La estructura de control de condicional *sí* (if)

6.3.1. El condicional if

Es posible tener programas en los que se deban cubrir diferentes casos, para los cuales se deberán retornar diferentes valores dadas unas condiciones.

La estructura de control condicional o de selección permite ejecutar, o un grupo de instrucciones u otro grupo si una condición se cumple o no, un condicional en C++ se especifica mediante el siguiente fragmento de código

```
if(<cond>){
  <body_1>
}else{
  <body_2>
};
```

donde se ejecutará `<body_1>` si `<cond>` se evalúa verdadero, en caso de que `<cond>` se evalúe falso de ejecutará `<body_2>`, después de ejecutar `<body_1>` o `<body_2>` se continua con la ejecución del resto del programa, después de la estructura `if`.

Ejemplo. *Valor absoluto de un número*

La función que permite calcular el valor absoluto de un número real es una función que recibe como parámetro de entrada un número real y retorna la distancia de ese valor al origen. La función valor absoluto en notación matemática se define como

$$\begin{aligned} valor\_absoluto : \mathbb{R} &\to \mathbb{R} \\ (x) &\mapsto \begin{cases} x, & \text{si } x \geq 0; \\ -x, & \text{en otro caso.} \end{cases} \end{aligned}$$

La codificación en C++ de esta función junto con su programa principal es

```
#include<iostream>
#include<cstdlib>

using namespace std;
```

```
double valor_absoluto(double x){
  double valor;
  if(x >= 0){
    valor = x;
  }else{
    valor = -x;
  };
  return valor;
};
```

```
int main(){
  double x;
  cout << "x? = ";
  cin >> x;
  cout << "El valor absoluto es: ";
  cout << valor_absoluto(x);
  cout << "\n";
  system("pause");
  return EXIT_SUCCESS;
};
```

Ejemplo. *El máximo entre dos números*

Una función que permite determinar el máximo de dos números reales, se puede definir como

$$maximo\_dos\_numeros : \mathbb{R} \times \mathbb{R} \longrightarrow \mathbb{R}$$

aquí se tienen dos casos, si el número a es mayor que b el valor máximo es a; en otro caso se debe retornar b. En notación matemática esto puede ser escrito de la siguiente forma

$$maximo\_dos\_numeros : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$$
$$(a, b) \mapsto \begin{cases} a, & \text{si } a > b; \\ b, & \text{en otro caso.} \end{cases}$$

La regla de traducción a función es similar a la anterior, sólo hay que tener en cuenta la instrucción condicional, y que si no se cumple la condición especificada en el `if`, se ejecutará el flujo de instrucciones especificado bajo el alcance del `else`

```
double maximo_dos_numeros(double a, double b){
  if(a > b){
    return a;
  }else{
    return b;
  };
};
```

6.3.2. El condicional `if` sin la sentencia `else`

En una estructura `if` la parte `else` es opcional, es decir, para el fragmento de código

```
if(<cond>){
  <body>
};
```

se ejecutará un grupo de instrucciones `<body>` si `<cond>` se evalúa verdadero, en otro caso salta al final de la estructura `if` y continua con el resto del programa, después de la estructura `if`.

Ejemplo. *El operador lógico "condicional"*

En los lenguajes de programación típicamente están definidos los operadores lógicos de la negación ($\neg$), la conjunción ($\wedge$) y la disyunción ($\vee$), pero el condicional y el bicondicional no lo están, por lo tanto si se quiere utilizar estos operadores es necesario construir las funciones que permitan utilizar estos operadores. Para el caso del condicional y a partir de la tabla de verdad para el operador condicional definido en el capítulo de lógica

$\xi(p)$	$\xi(q)$	$\xi(p \to q)$
V	V	V
V	F	F
F	V	V
F	F	V

se puede definir una función que permite calcular la operación condicional de un par de variables booleanas y que retorna el resultado de operar los valores mediante un condicional, de la siguiente manera

$$condicional : \mathbb{B} \times \mathbb{B} \longrightarrow \mathbb{B}$$

aquí se tienen dos casos, primero, si el antecedente es verdadero y el consecuente es falso, entonces el resultado de aplicar el condicional el falso, para cualquier otro caso el condicional es verdadero. En notación matemática esto puede ser escrito de la siguiente manera

$$condicional : \mathbb{B} \times \mathbb{B} \to \mathbb{B}$$
$$(p, q) \mapsto \begin{cases} F, & \text{si } (p = V) \wedge (q = F); \\ V, & \text{en cualquier otro caso.} \end{cases}$$

Una posible codificación en C++ de esta función seria

```
bool condicional(bool p, bool q){
  if(p == true && q == false){
    return false;
  }else{
    return true;
  };
};
```

obsérvese que es posible construir una función que utilice sólo una estructura `if` sin la sentencia `else` que es mucho más sencilla que la función presentada anteriormente,

```
bool condicional(bool p, bool q){
  if(p){
    return q;
  };
  return true;
};
```

en esta función se tiene en cuenta que si la premisa `p` tiene valor `true`, entonces el resultado está dado por el valor de la conclusión `q`, y si el valor de `p` es `false`, entonces el condicional tendrá como valor `true`.

6.3.3. Estructuras `if` enlazadas

Otra de las opciones para utilizar una estructura `if` es la de enlazar varias estructuras `if`, de tal manera que solamente se pueda ejecutar un grupo de instrucciones dependiendo de cual de las opciones se evalúa verdadero. De la misma manera que en el caso anterior la parte `else` final es opcional. La codificación en C++ de las estructuras `if` enlazadas es la siguiente

```
if(<cond_1>){
  <body_1>
}else if(<cond_2>){
  <body_2>
}
  ...
}else if(<cond_i>){
  <body_i>
}
  ...
}else if(<cond_n-1>){
  <body_n-1>
}else{
  <body_n>
};
```

donde se ejecutará `<body_1>` si `<cond_1>` se evalúa verdadero, en caso de que `<cond_1>` se evalúe falso de ejecutará `<body_2>` si `<cond_2>` se evalúa verdadero, y así se continuará revisando cada una de las condiciones si la anterior se evalúa falso. Si algún `<cond_i>` se evalúa verdadero se ejecuta su respectivo `<body_i>` y después de ejecutar todas las instrucciones del `<body_i>` se continua con el resto del programa, después de la estructuras `if` enlazadas.

Ejemplo. *El descuento del día*

Una tienda tiene las siguientes promociones

> Si lleva más de 5 productos del mismo tipo le realizan un descuento del 5 %. Si lleva más de 10 el 10 %. Si lleva más de 20 el 20 % de descuento. Realizar un programa que dado el número de productos y el precio del producto determine el valor a pagar por el cliente.

La siguiente función permitirá calcular el valor deseado

$$descuento(precio, n) = valor$$

Si se establecen las variables:

$$\begin{aligned} precio &:= \text{Valor de cada producto} \\ n &:= \text{Número de productos} \\ valor &:= \text{Valor total a pagar despues de aplicar el descuento} \end{aligned}$$

entonces

$$descuento : \mathbb{R} \times \mathbb{Z} \to \mathbb{R}$$

$$(precio, n) \mapsto \begin{cases} n * precio, & n \leq 5; \\ n * precio * 0.95, & 5 < n \leq 10; \\ n * precio * 0.90, & 10 < n \leq 20; \\ n * precio * 0.80, & \text{en otro caso.} \end{cases}$$

La codificación en C++ de esta función es

```
double descuento(double precio, int n){
  if(n <= 5){
    return n * precio;
  }else if(5 < n && n <= 10){
    return n * precio * 0.95;
  }else if(10 < n && n <= 20){
    return n * precio * 0.90;
  }else{
    return n * precio * 0.80;
  };
};
```

Otra posible escritura de la función puede ser

```
double descuento(double precio, int n){
  double valor;
  if(n <= 5){
    valor = n * precio;
  }else if(5 < n && n <= 10){
    valor = n * precio * 0.95;
  }else if(10 < n && n <= 20){
    valor = n * precio * 0.90;
  }else{
    valor = n * precio * 0.80;
  };
  return valor;
};
```

6.4. Validación de datos usando condicionales

Teniendo en cuenta que el algoritmo para el cálculo del área de un rectángulo dada la definición del tipo de dato podrían estarse leyendo largos o anchos negativos. La notación matemática también permite restringir el dominio y el rango de los conjuntos quedando la función de la siguiente forma

$$area\_rectangulo : \mathbb{R}^{0,+} \times \mathbb{R}^{0,+} \to \mathbb{R}^{0,+}$$
$$(l, a) \mapsto l * a$$

Dicha validación es suficiente a nivel de notación matemática, pero en programación dichas validaciones sobre los parámetros de entrada corresponderán al programa principal. Para realizar estas validaciones se tienen dos opciones, el uso de un condicional ó el uso de un ciclo (este último se explicará en el capitulo de ciclos).

En este caso si el usuario ingresa un largo o un ancho negativo, se suspenderá la ejecución del programa mostrando un mensaje de error retornando `EXIT_FAILURE` al sistema operativo.

La codificación en C++ de la función para calcular el área de un rectángulo haciendo la validación tanto del ancho como de la altura es la siguiente

```
#include<iostream>
#include<cstdlib>

using namespace std;
```

```
double area_rectangulo(double l, double a){
  return l * a;
};
```

```
int main(){
  double largo;
  double ancho;
  cout << "largo? = ";
  cin >> largo;
  if(largo < 0){
    cout << "El largo no es valido";
    cout << "\n";
    system("pause");
    return EXIT_FAILURE;
  };
  cout << "ancho? = ";
  cin >> ancho;
  if(ancho < 0){
    cout << "El ancho no es valido";
    cout << "\n";
    system("pause");
    return EXIT_FAILURE;
  };
  cout << "El area del rectangulo es: ";
  cout << area_rectangulo(largo, ancho);
  cout << "\n";
  system("pause");
  return EXIT_SUCCESS;
};
```

6.5. Ejercicios

1. Hacer un programa que dado el área del corral de unas gallinas y el número de gallinas en el corral determine el área que le corresponde a cada gallina.

2. Construir una función que dados tres números reales calcule el máximo de los tres.

3. Dadas las longitudes de los dos brazos de una palanca y el peso puesto en el brazo más largo de la palanca, calcular el peso que se puede poner en el brazo más corto para que la palanca quede en equilibrio.

4. Dadas las longitudes de los dos brazos de una palanca y el peso puesto en el brazo más corto de la palanca, calcular el peso que se puede poner en el brazo más largo para que la palanca quede en equilibrio.

5. Dados los pesos que se pueden poner en cada uno de los lados de la palanca y la longitud total de la palanca determinar la longitud del brazo mas largo para que la palanca quede en equilibrio.

6. Dados los pesos que se pueden poner en cada uno de los lados de la palanca y la longitud total de la palanca determinar la longitud del brazo mas corto para que la palanca quede en equilibrio.

7. Para crear un litro del compuesto D, se requiere que su composición por partes esté conformada de la siguiente manera, 5 partes deben ser del producto A, 8 partes deben ser del producto B y 7 partes deben ser del producto C. Si se requiere crear 10 litros del compuesto D. ¿Cuántos litros del producto A se requieren?, ¿Cuántos litros del producto B se requieren?, ¿Cuántos litros del producto C se requieren?.

8. Si se tienen x litros de A, y litros de B y z litros de C. ¿Cuántos litros del compuesto D se pueden obtener?.

9. En la convención republicana, se reúnen 100 personas de las cuales x son mujeres e y son hombres. Si en la convención se dispone de z mesas y se dispone que la diferencia entre número de hombres y mujeres en cada mesa no debe superar 2. ¿Cuantas sillas para mujeres se deben poner por mesa?, ¿Cuantas sillas por hombre se deben poner?. Si los votos de las mujeres valen 1.5 veces los votos del hombre. ¿Cuántas mujeres deberían estar en la convención para que el candidato sea elegido por votos de solo mujeres? Si los votos de los hombres valen 1.5 veces los votos de las mujeres. ¿Cuántas mujeres deberían estar en la convención para que el candidato sea elegido por votos de solo mujeres?

10. Dado el centro y el radio de un círculo, determinar si un punto pertenece o no al círculo.

11. Dadas tres longitudes positivas, determinar si con esas longitudes se puede construir un triángulo.

Capítulo 7

Funciones recursivas

Definición. En 1952 Stephen Kleene definió formalmente en [Kleene 1952] que una "función parcial recursiva" de enteros no negativos es cualquier función f definida por un sistema no contradictorio de ecuaciones de las cuales las partes derechas e izquierdas están compuestas a partir de:

i. Símbolos funcionales (por ejemplo, f, g, h, etc.),

ii. Variables para enteros no negativos (por ejemplo, x, y, z, etc.),

iii. La constante 0, y

iv. La función primitiva sucesor $s(x) = x + 1$.

Ejemplo. El siguiente es un sistema que define la función parcial recursiva $f(x, y)$ que permite computar el producto de x con y.

$$\begin{aligned} f(x, 0) &= 0 \\ f\big(x, s(y)\big) &= g\big(f(x, y), x\big) \\ g(x, 0) &= x \\ g\big(x, s(y)\big) &= s\big(g(x, y)\big) \end{aligned}$$

Nótese que las ecuaciones podrían no determinar el valor de f para cada posible entrada, y que en ese sentido la definición es lo que se definió como función parcial. Si el sistema de ecuaciones determina el valor de f para cada entrada, entonces la definición es lo que se definió como función total. Cuando se usa el término función recursiva, en este caso se esta hablando de forma implícita de que la función recursiva es total.

El conjunto de funciones que pueden ser definidas recursivamente en esta forma se sabe que son equivalente a el conjunto de funciones computables por una máquina de Turing o por medio del lambda calculo.

Ya que este libro es un texto introductorio a la programación, no se tratarán las funciones recursivas con todo su detalle formal ya que esto está mucho más allá del alcance de este libro; en cambio se intentará caracterizar más concretamente las funciones recursivas que usualmente son utilizadas a un nivel introductorio de la programación, mediante la

siguiente definición debilitada de función recursiva. Hay que tener en mente que no se pretende que esta caracterización sea exhaustiva con respecto al conjunto de todas las funciones recursivas.

Definición (Definición débil de función recursiva)**.**
Una función $f : A \to B$ se dice *recursiva* si y sólo si f está definida por casos (mediante un predicado sobre los argumentos), en donde al menos uno de los casos se define usando la misma función f y los argumentos, y al menos uno de los otros casos se define usando solamente los argumentos sin involucrar la función f.

Mediante el uso de funciones recursivas se puede solucionar cualquier problema que es potencialmente solucionable haciendo uso de un computador.

A continuación se presentan algunos ejemplos de problemas clásicos que se pueden solucionar mediante el uso de funciones recursivas, haciendo énfasis en la metodológica que de debe seguir para identificar la regla recursiva que se encuentra implícita en cada uno de los problemas.

Ejemplo. *Potencia de un número*

En este ejemplo se definirá una función recursiva que permita hallar un número real elevado a un número natural. Para expresar una función que calculé esta operación, en primera instancia se construye la expresión $potencia : \mathbb{R} \times \mathbb{N} \to \mathbb{R}$ que define la función que tiene como entrada un número real que representa la base y un número natural que indica el exponente, y como salida se obtendrá un número real que será la potencia. Por facilidad, aquí se asumirá que $0^0 = 1$.

Ahora observesé que en general si se tiene una base b y un exponente n, entonces por definición

$$b^n = \underbrace{b * b * b * \cdots * b * b}_{n\text{–veces}}$$

si se usa la propiedad asociativa del producto de números reales, se tiene que

$$b^n = \underbrace{b * b * b * \cdots * b * b}_{n\text{–veces}} = \underbrace{(b * b * \cdots * b * b)}_{n-1\text{–veces}} * b$$

lo que es equivalente a

$$b^n = \underbrace{b * b * b * \cdots * b * b}_{n\text{–veces}} = \underbrace{(b * b * \cdots * b * b)}_{n-1\text{–veces}} * b = b^{n-1} * b$$

A partir de esta observación se puede dar una definición recursiva usando funciones. La declaración de esta función junto con su cuerpo se hará de la siguiente manera

$$potencia(b, n) = p$$

Si se establecen las variables:

$$b := \text{Base}$$
$$n := \text{Exponente}$$
$$p := \text{Potencia } b^n$$

entonces

$$potencia : \mathbb{R} \times \mathbb{N} \to \mathbb{R}$$
$$(b, n) \mapsto \begin{cases} 1, & \text{si} \quad n = 0; \\ potencia(b, n-1) * b, & \text{en otro caso.} \end{cases}$$

La codificación en C++ de esta función es

```
#include<iostream>
#include<cstdlib>

using namespace std;
```

```
double potencia(double b, int n){
  if(n == 0){
    return 1;
  };
  return  potencia(b,n - 1) * b;
};
```

```
int main(){
  double b;
  int n;
  cout << "b? = ";
  cin >> b;
  cout << "n? = ";
  cin >> n;
  cout << "potencia(b,n) = ";
  cout << potencia(b,n);
  cout << "\n";
  system("pause");
  return EXIT_SUCCESS;
};
```

Ejemplo. *Pago del interés compuesto mes vencido*

Supónga que solicita un préstamo de $1'000.000 durante un año, el prestamista cobra un interés del 5 % mensual mediante la modalidad de interes compuesto mes vencido. ¿Cuál es el total del dinero que debe pagar cuando ha transcurrido el año por el cual solicitó el prestamo?.

Para calcular el valor solicitado hay que observar que:

- Para cero meses se tiene que hay que pagar

$$\$1'000.000$$

pues no ha transcurrido ningún mes.

- Para un mes se tiene que hay que pagar

$$\$1'000.000 + \$1'000.000 * 0.05 = [\$1'000.000](1 + 0.05)$$

lo prestado más los intereses de un mes.

- Para dos meses se tiene que hay que pagar

$$[\$1'000.000](1 + 0.05) + [\$1'000.000](1 + 0.05)0.05 = \Big[[\$1'000.000](1 + 0.05)\Big](1 + 0.05)$$

lo que se debía pagar en el mes anterior más los intereses de esa cantidad.

A partir de las observaciones anteriores, ya se detecta la regla recursiva con la que se puede calcular el interés compuesto mes vencido en general, con lo cual se puede diseñar una función recursiva que permita calcular el valor total a pagar para cualquier monto, cualquier interés y cualquier intervalo de tiempo.

$$pago(m, i, n) = valor$$

donde se tienen las variables

$$\begin{aligned} m &:= \text{Cantidad de dinero solicitado como prestamo} \\ i &:= \text{Interes} \\ n &:= \text{Número de meses por el cual se solicita el pretamo} \\ valor &:= \text{Valor total a pagar por el prestamo de la cantidad } m \\ &\quad \text{por } n \text{ meses con un interés } i \text{ utilizando el método de} \\ &\quad \text{interés compuesto mes vencido} \end{aligned}$$

entonces

$$pago : \mathbb{R}^+ \times \mathbb{R}^+ \times \mathbb{Z}^+ \to \mathbb{R}^+$$

$$(m, i, n) \mapsto \begin{cases} m, & n = 0; \\ pago(m, i, n-1) * (1 + i), & \text{en otro caso.} \end{cases}$$

```
#include<iostream>
#include<cstdlib>

using namespace std;
```

```
double pago(double m, double i, int n){
  if(n == 0){
    return m;
  };
  return pago(m,i,n - 1) * (1 + i);
};
```

```
int main(){
  double m;
  double i;
  int n;
  cout << "m? = ";
  cin >> m;
  cout << "i? = ";
  cin >> i;
  cout << "n? = ";
  cin >> n;
  cout << "pago(m,i,n) = ";
  cout << pago(m,i,n);
  cout << "\n";
  system("pause");
  return EXIT_SUCCESS;
};
```

Ejemplo. *Número de listas de los elementos de un conjunto*

Suponga que selecciona cuatro cartas distintas de una baraja de poker, que se van a representar por los símbolos

si con estas cartas se forma el conjunto $cartas = \{\clubsuit, \blacklozenge, \heartsuit, \spadesuit\}$. ¿De cuántas formas distintas se pueden organizar las cartas?

Como se van a listar todas las formas posibles en que se pueden organizar las cartas, el orden si importa. Una estrategia para encontrar el número de listas puede ser el siguiente:

1. Se selecciona una carta del conjunto *cartas* de forma arbitraria pero fija, por ejemplo la carta $\blacklozenge$.

2. Ya fijada la carta $\blacklozenge$, el resto del trabajo consiste en hallar el número de formas distintas de organizar las cartas restantes, es decir, el conjunto $cartas \setminus \{\blacklozenge\} = \{\clubsuit, \heartsuit, \spadesuit\}$.

3. Ahora por ejemplo se selecciona la carta $\spadesuit$ de forma arbitraria pero fija.

4. A continuación, el trabajo se reduce a hallar el número de formas distintas de organizar las cartas restantes, es decir, el conjunto $cartas \setminus \{\blacklozenge, \spadesuit\} = \{\clubsuit, \heartsuit\}$.

5. Posteriormente, por ejemplo se puede seleccionar de forma arbitraria pero fija la carta ♥.

6. Para finalizar, el trabajo se reduce a hallar el número de formas distintas de organizar las cartas restantes, es decir el conjunto $cartas \setminus \{♦, ♠, ♥\} = \{♣\}$. Como para este conjunto sólo se tiene una opción, entonces el número de formas distintas de organizar un conjunto de una carta es 1.

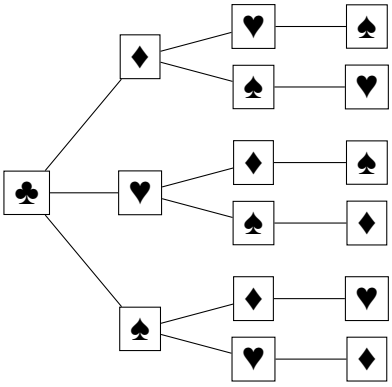
Siguiendo los pasos anteriores, se obtuvo la lista

♦ ♠ ♥ ♣

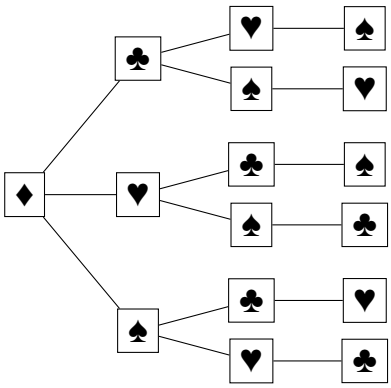
Como la selección de las cartas se hizo de forma arbitraria, entonces, para poder listar todos los posibles ordenamientos, se tiene que el paso del numeral 1 se puede realizar de cuatro formas. Por cada una de estas escogéncias se hace la selección de una carta de un conjunto con un elemento menos, como ocurre en el paso del numeral 3; esto se puede realizar de tres formas posibles. Por cada par de escogéncias se hace la selección de una carta de un conjunto con dos elementos menos, como ocurre en el paso del numeral 5; esto se puede realizar de dos formas posibles. Por cada trío de escogéncias se hace la selección de una carta de un conjunto con tres elementos menos. Para este caso el conjunto restante tiene un solo elemento y por lo tanto sólo hay una posible selección. De lo anterior se concluye que el número de formas de listar los elementos de un conjunto con cuatro elementos es

$$4 * 3 * 2 * 1 = 24$$

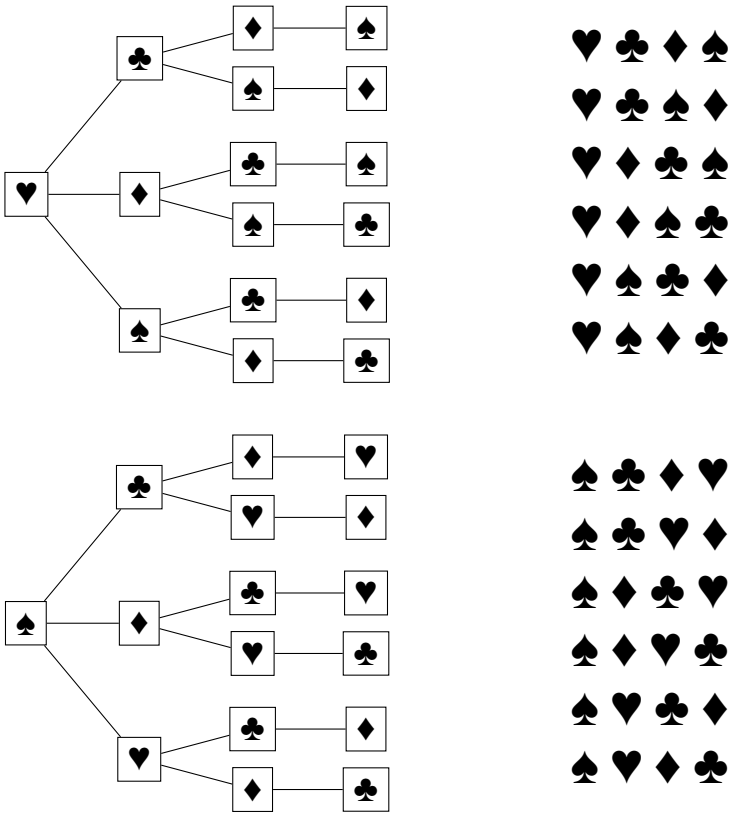
En los diagramas que se presentan a continuación se exhibe la forma sistemática en que se pueden obtener todas la posibles listas que se forman con las cartas ♣, ♦, ♥, ♠.



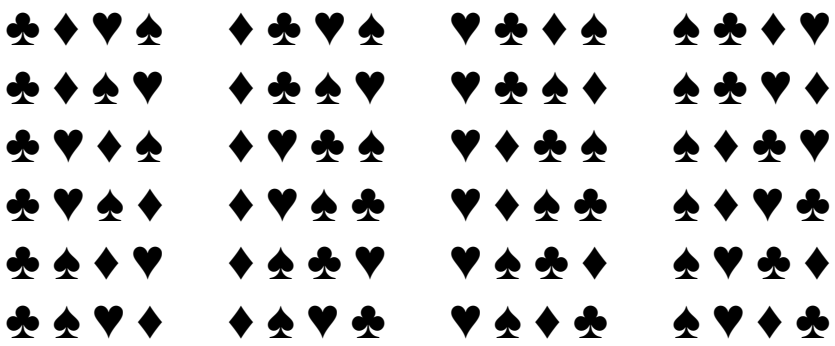
♣ ♦ ♥ ♠
♣ ♦ ♠ ♥
♣ ♥ ♦ ♠
♣ ♥ ♠ ♦
♣ ♠ ♦ ♥
♣ ♠ ♥ ♦



♦ ♣ ♥ ♠
♦ ♣ ♠ ♥
♦ ♥ ♣ ♠
♦ ♥ ♠ ♣
♦ ♠ ♣ ♥
♦ ♠ ♥ ♣



El listado de las 24 posibles formas en que se pueden organizar las cuatro cartas es el siguiente



En general, para un conjunto A con cardinal $|A| = n$, se tiene que el número de formas de listar todas lo formas en que se pueden organizar los elementos de A es

$$n * (n-1) * (n-2) * \cdots * 3 * 2 * 1$$

este valor depende solamente de n, es una función, se denota por el símbolo $n!$ y se llama es factorial del número n. Para el caso del conjunto $\varnothing$, se puede demostrar que $0! = 1$.

$$n! = n * (n-1) * (n-2) * \cdots * 3 * 2 * 1$$

A partir de las observaciones anteriores se puede obtener la función recursiva factorial $n!$, distinta a la exhibida anteriormente

$$fact(n) = f$$

Si se establecen las variables:

$$n := \text{Número al cual se le va a calcular el factorial}$$
$$f := \text{Factorial de } n$$

entonces

$$fact : \mathbb{N} \to \mathbb{N}$$
$$(n) \mapsto \begin{cases} 1, & \text{si} \quad n = 0; \\ n * fact(n-1), & \text{en otro caso.} \end{cases}$$

La codificación en C++ de esta función junto con su programa principal es

```
#include<iostream>
#include<cstdlib>

using namespace std;
```

```
int fact(int n){
  if(n == 0){
    return 1;
  };
  return n * fact(n - 1);
};
```

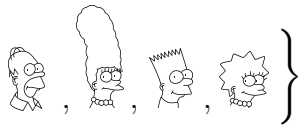
```
int main(){
  int n;
  cout << "n? = ";
  cin >> n;
  cout << "fact(n) = ";
  cout << fact(n);
  cout << "\n";
  system("pause");
  return EXIT_SUCCESS;
};
```

Ejemplo. *Conteo de subconjuntos*

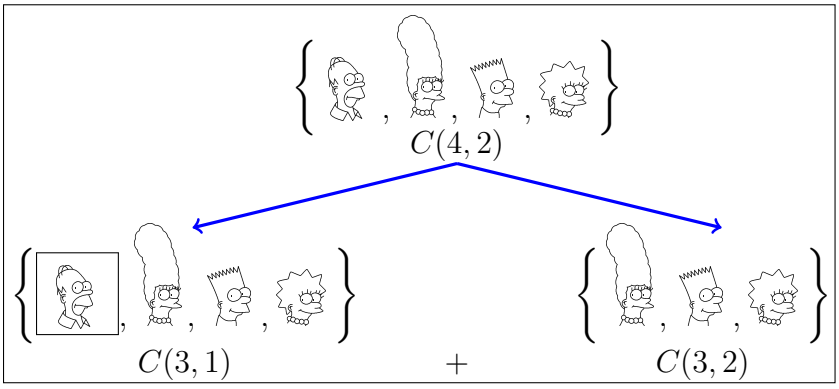
Los Simpsons van a un parque de diversiones y quieren subir a la montaña rusa, por lo que sólo pueden subir Homero, Marge, Bart y Lisa, y en dicha montaña rusa cada vagón sólo dispone de dos puestos. ¿De cuantas formas se pueden formar parejas de la familia Simpson para que suban al vagón de la montaña rusa?.

Como se van a formar parejas, el orden no importa. Una estrategia para encontrar el número de parejas puede ser el siguiente:

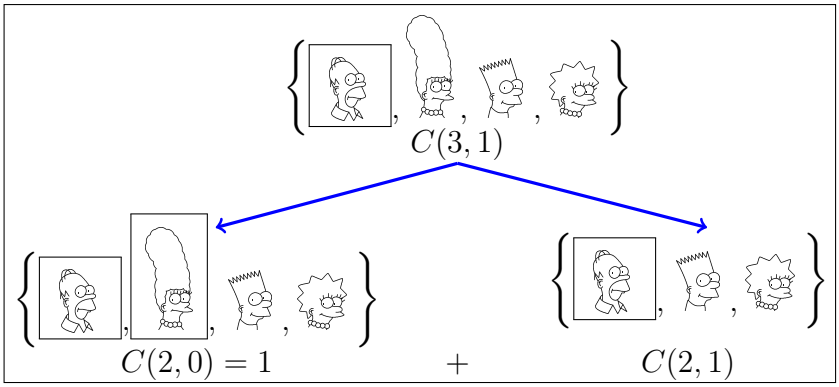
1. Se listan los elementos del conjunto, en este caso podría ser

$$Simpsons = \left\{ \text{  } \right\}$$

2. Dado el conjunto *Simpsons*, para la pareja que se va seleccionar se puede escoger o no a Homero, como se observa a continuación

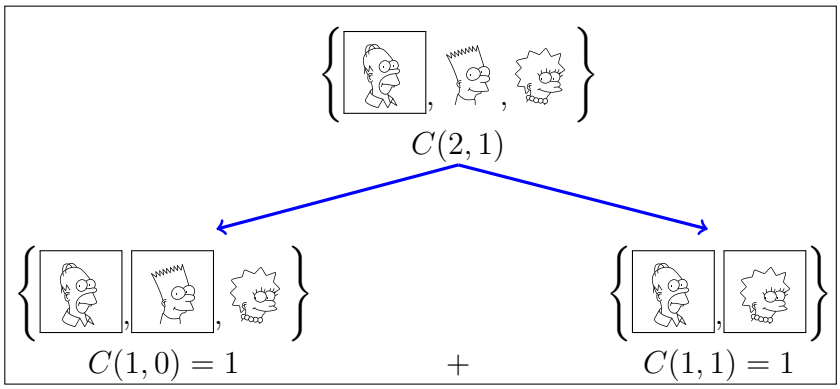


3. Si en el numeral 2 se seleccionó a Homero, entonces ahora se puede escoger o no a Marge, como se observa a continuación



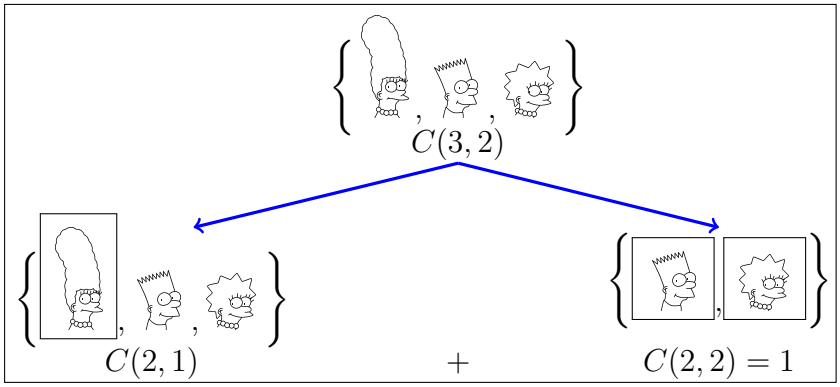
si se escoge a Marge, entonces ya se tiene una pareja.

4. Si en el numeral 2 se seleccionó a Homero y en el 3 no se seleccionó a Marge, entonces ahora se puede escoger o no a Bart, como se observa a continuación



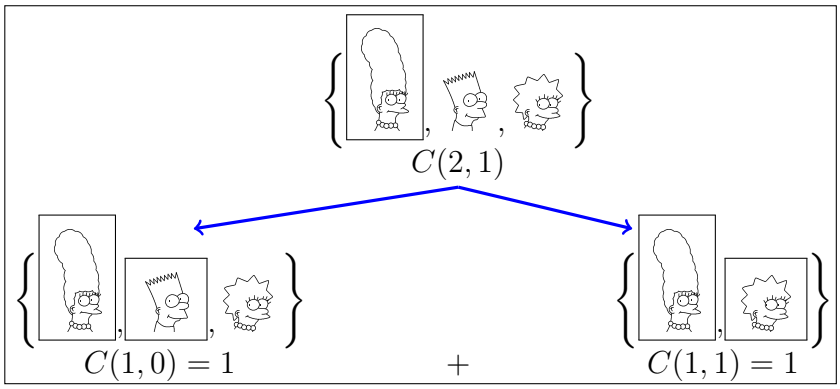
si se escoge a Bart, entonces ya se tiene otra pareja, si no entonces Lisa debe hacer parte de la siguiente pareja.

5. Si en el numeral 2 no se seleccionó a Homero, entonces ahora se puede escoger o no a Marge, como se observa a continuación



si se no escoge a Marge, entonces ya se tiene una nueva pareja.

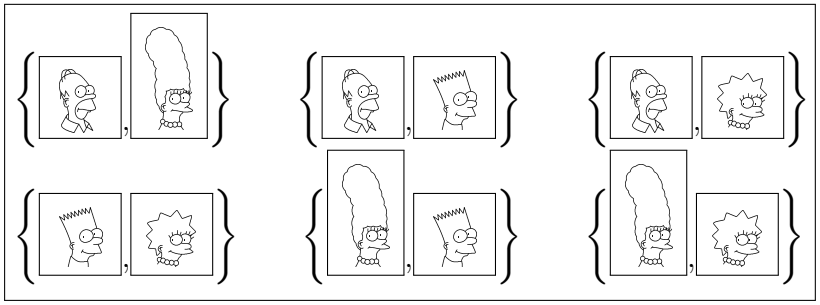
6. Si en el numeral 5 se seleccionó a Marge, entonces ahora se puede escoger o no a Bart, como se observa a continuación



si se escoge a Bart, entonces ya se tiene otra pareja, si no entonces Lisa debe hacer parte de la última pareja.

7. Después de hacer el conteo exhaustivo de la parejas que se pueden formar, se observa que existen 6 parejas que contienen dos miembros de la familia de los *Simpsons*, estas son:

$$\begin{aligned}
C(4,2) &= C(3,1) + C(3,2) \\
&= \big(C(2,0) + C(2,1)\big) + C(3,2) \\
&= \big(1 + \big(C(1,0) + C(1,1)\big)\big) + C(3,2) \\
&= \big(1 + \big(1 + 1\big)\big) + C(3,2) \\
&= \big(1 + 2\big) + C(3,2) \\
&= 3 + C(3,2) \\
&= 3 + \big(C(2,1) + C(2,2)\big) \\
&= 3 + \big(\big(C(1,0) + C(1,1)\big) + 1\big) \\
&= 3 + \big(\big(1 + 1\big) + 1\big) \\
&= 3 + (2 + 1) = 3 + 3 = 6
\end{aligned}$$



A partir del ejemplo anterior, si la función

$$C(n,k) = c$$

representa el número de subconjuntos de k elementos de un conjunto de n elementos, ahora se puede construir una función recursiva que permita hacer el conteo del número de estos subconjuntos, de esta forma

Si se establecen las variables:

$n :=$ Número de elementos del conjunto
$k :=$ Número de elementos de los subconjuntos
$c :=$ Número de subconjuntos de k elementos de un conjunto de n elementos

entonces

$$C : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$$
$$(n,k) \mapsto \begin{cases} 0, & \text{si} \quad k > n; \\ 1, & \text{si} \quad (k = 0) \vee (n = k); \\ C(n-1,k-1) + C(n-1,k), & \text{en otro caso.} \end{cases}$$

La codificación en C++ de esta función junto con su programa principal es

```
#include<iostream>
#include<cstdlib>

using namespace std;
```

```
int C(int n, int k){
  if(k > n){
    return 0;
  };
  if(k == 0 || n == k){
    return 1;
  };
  return C(n - 1,k - 1) + C(n - 1,k);
};
```

```
int main(){
  int n;
  int k;
  cout << "n? = ";
  cin >> n;
  cout << "k? = ";
  cin >> k;
  cout << "C(n,k) = ";
  cout << C(n,k);
  cout << "\n";
  system("pause");
  return EXIT_SUCCESS;
};
```

Ejemplo. *Los conejos y los números de Fibonacci*

Una pareja de conejos recién nacidos (uno de cada sexo) se liberan en una isla. Los conejos no pueden tener descendencia hasta que cumplen dos meses. Una vez que cumplen dos meses, cada pareja de conejos tiene como descendencia otra pareja de conejos cada mes[1]. ¿Cuál es la cantidad de parejas de conejos en la isla una vez transcurrido un año, suponiendo que ningún conejo muere?.

[1]Este problema fué propuesto originalmente por el italiano Leonardo Pisano Bigollo (1170–1250), más conocido como Leonardo de Pisa o Fibonacci (que significa hijo de Bonacci, *filius Bonacci*) en su libro *Liber abaci* publicado en 1202.

Si f_n denota la cantidad de parejas de conejos en el mes n, entonces, en el mes cero, en éste aun no se ha hecho la liberación de la pareja de conejos, por lo tanto la cantidad de parejas es $f_0 = 0$.

$n = 0,\ f_0 = 0$

Durante el primer mes, en este se hace la liberación de la primera pareja de conejos, pero aún no han alcanzado la edad para reproducirse, por lo tanto, no ha habido descendencia, por lo tanto, $f_1 = 1$.

$n = 1,\ f_1 = 1$ 

Durante el segundo mes, ya había una pareja de conejos del mes anterior y éstos aún no han alcanzado la edad para reproducirse, por lo tanto, no hubo descendencia, de donde f_2 es igual a la cantidad de conejos que habían en el mes anterior más la descendencia que produjeron las parejas de más de dos meses, es decir, $f_2 = 1$.

$n = 2,\ f_2 = f_1 + f_0 = 1 + 0 = 1$ 

Durante el tercer mes, ya había una pareja de conejos del mes anterior y durante el transcurso de este mismo mes los conejos alcanzaron la madures para reproducirse, por lo tanto hubo descendencia, de donde f_3 es igual a la cantidad de conejos del mes anterior más la descendencia que se produjo en este mes, es decir, $f_3 = 2$.

$n = 3,\ f_3 = f_2 + f_1 = 1 + 1 = 2$ 

Durante el cuarto mes ya habían dos parejas de conejos del mes anterior, y la pareja madura es la que había en el segundo mes, por lo tanto, la descendencia fue generada sólo por esa pareja, de donde f_4 es igual a la cantidad de parejas del mes anterior más la descendencia que generé la pareja del segundo mes, es decir, $f_4 = f_3 + f_2 = 2 + 1 = 3$.

$n = 4,\ f_4 = f_3 + f_2 = 2 + 1 = 3$ 

Durante el quinto mes ya habían tres parejas de conejos del mes anterior, y de éstas hay dos parejas maduras, que son las que habían en el tercer mes, por lo tanto, la descendencia fue generada por esas dos parejas, de donde f_5 es igual a la cantidad de parejas del mes anterior más la descendencia que generen las parejas del tercer mes, es decir, $f_5 = f_4 + f_3 = 3 + 2 = 5$.

$n = 5,\ f_5 = f_4 + f_3 = 3 + 2 = 5$

Haciendo análisis similares se obtienen los siguientes resultados:

$$\begin{aligned}
&\text{Para } n = 6, \quad \text{se tiene que} \quad f_6 = f_5 + f_4 = 5 + 3 = 8\\
&\text{Para } n = 7, \quad \text{se tiene que} \quad f_7 = f_6 + f_5 = 8 + 5 = 13\\
&\text{Para } n = 8, \quad \text{se tiene que} \quad f_8 = f_7 + f_6 = 13 + 8 = 21\\
&\text{Para } n = 9, \quad \text{se tiene que} \quad f_9 = f_8 + f_7 = 21 + 13 = 34\\
&\text{Para } n = 10, \quad \text{se tiene que} \quad f_{10} = f_9 + f_8 = 34 + 21 = 55\\
&\text{Para } n = 11, \quad \text{se tiene que} \quad f_{11} = f_{10} + f_9 = 55 + 34 = 89\\
&\text{Para } n = 12, \quad \text{se tiene que} \quad f_{12} = f_{11} + f_{10} = 89 + 55 = 144
\end{aligned}$$

De aquí que, transcurrido el primer año, en la isla habrán 144 parejas de conejos.

A los números que son generados utilizando esta regla se les conoce como números de Fibonacci.

A partir del análisis anterior, se puede diseñar una función recursiva que permite calcular cualquier número de Fibonacci.

$$fibo(n) = f$$

donde se tienen las variables

$n :=$ Número del cual se desea calcular su número de Fibonacci
$f :=$ Número de Fibonacci de n

entonces

$$fibo : \mathbb{N} \to \mathbb{N}$$
$$(n) \mapsto \begin{cases} 0, & \text{si} \quad n = 0; \\ 1, & \text{si} \quad n = 1; \\ fibo(n-1) + fibo(n-2), & \text{en otro caso.} \end{cases}$$

```
#include<iostream>
#include<cstdlib>

using namespace std;
```

```
int fibo(int n){
  if(n == 0){
    return 0;
  }else if(n == 1){
    return 1;
  };
  return fibo(n - 1) + fibo(n - 2);
};
```

```
int main(){
  int n;
  cout << "n? = ";
  cin >> n;
  cout << "Fibonacci(n) = ";
  cout << fibo(n);
  cout << "\n";
  system("pause");
  return EXIT_SUCCESS;
};
```

Ejemplo. *Número primo*

Determinar si un número mayor a 1 es primo o no (sólo es divisible por 1 y por él mismo).

Por definición, n es primo si no es compuesto.

Un número m es *compuesto* si se puede descomponer en la forma

$$m = p * q$$

donde $p, q \in \mathbb{N}$, $1 < p < m$ y $1 < q < m$. Entonces, para saber si un número es primo es equivalente a verificar que no es compuesto, es decir, que no hay un número $k \in \mathbb{N}$, tal que $1 < k < m$ y que k sea divisor de m.

De lo anterior, para saber si un número n es compuesto hay que ir probando con los números desde 2 hasta $n - 1$, y observar si alguno de ellos es divisor de n; si no es así, entonces n es primo.

Una observación adicional que hará más eficiente el algoritmo es la que se puede concluir a partir del siguiente teorema

Teorema. *Si un número m es compuesto, entonces existe $k \in \mathbb{N}$ tal que k es divisor de m y*

$$1 < k \leq \sqrt{m}.$$

Demostración. Como m es un número compuesto entonces m es de la forma $m = p * q$, con $p > 1$ y $q > 1$. A partir de p y q se tiene que $p \leq \sqrt{m}$ o $q \leq \sqrt{m}$, pues si no fuese es así, entonces, se tendría lo contrario, es decir, $p > \sqrt{m}$ y $q > \sqrt{m}$, si esto se tuviera, entonces por construcción,

$$m = p * q > (\sqrt{m})^2 = m,$$

lo cual es una contradicción, por lo tanto, tener que$p > \sqrt{m}$ y $q > \sqrt{m}$ genera una contradicción, de donde la afirmación que se cumple es que $p \leq \sqrt{m}$ o $q \leq \sqrt{m}$; ese valor p o q que es menor o igual a $\sqrt{m}$ resulta ser el valor k que es el divisor de m que cumple con la desigualdad. ☑

A partir del teorema anterior, se pueden diseñar las funciones que permiten determinar si un número es primo o no. Inicialmente es necesario construir una función auxiliar o ayudante (en inglés *helper*) que nos permita responder a la pregunta ¿Dado un par de enteros positivos n y d, n es múltiplo de algún valor entre d y $\sqrt{n}$ inclusive?. La siguiente función recursiva permite responder a la pregunta anterior.

$$multiplo(n, d) = valor$$

donde se establecen las variables

$$\begin{aligned} n &:= \text{Número del cual se desea saber si es multiplo de } d \\ d &:= \text{Número candidato a ser divisor de } n \\ valor &:= true \text{ si } d \text{ es multiplo de } n \text{ y } false \text{ si no lo es} \end{aligned}$$

entonces

$$multiplo : \mathbb{N} \times \mathbb{P} \to \mathbb{B}$$
$$(n, d) \mapsto \begin{cases} V, & \text{si} \quad n \text{ mód } d = 0; \\ F, & \text{si} \quad d > \sqrt{n}; \\ multiplo(n, d+1), & \text{en otro caso.} \end{cases}$$

Con el uso de la anterior función se puede diseñar una nueva función que permita determinar si un entero mayor que 1 es primo o no, la idea es probar los números entre 2 y $\sqrt{n}$ usando la función anterior y verificar si n es múltiplo de alguno de estos números, si se da este caso, entonces el número no es primo, en caso contrario se tendrá que el número es primo. La siguiente función permite determinar si un número es primo o no

$$primo(n) = valor$$

donde se tienen las variables

$$\begin{aligned} n &:= \text{Número del cual se desea establecer si es primo o no} \\ valor &:= true \text{ si } n \text{ es primo y } false \text{ si no lo es} \end{aligned}$$

entonces

$$primo : \mathbb{N} \setminus \{0, 1\} \to \mathbb{B}$$
$$(n) \mapsto \neg multiplo(n, 2)$$

La codificación en C++ de estas funciones es la siguiente, aquí se utiliza la función `sqrt(x)`, la cual está previamente construida en la librería `cmath`, ésta se puede incluir

mediante la instrucción `#include<cmath>`. En los ejercicios se solicitará construir la función $\left[\sqrt{x}\right]$ y en capítulo de ciclos se explicará como construir la función $\sqrt{x}$ en general:

```
#include<iostream>
#include<cstdlib>
#include<cmath>

using namespace std;
```

```
bool multiplo(int n, int d){
  if(n % d == 0){
    return true;
  };
  if(d > sqrt(n)){
    return false;
  };
  return multiplo(n,d + 1);
};
```

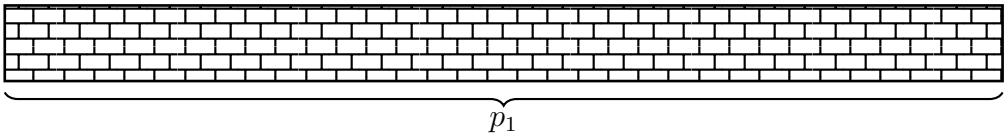
```
bool primo(int n){
  return !multiplo(n,2);
};
```

```
int main(){
  int n;
  cout << "n? = ";
  cin >> n;
  cout << "Es n primo? = ";
  cout << primo(n);
  cout << "\n";
  system("pause");
  return EXIT_SUCCESS;
};
```

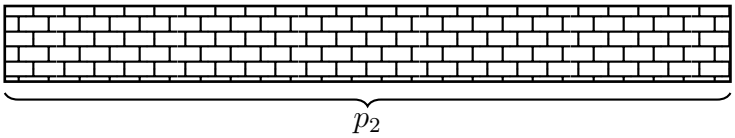
Ejemplo. *El mural de una empresa*

Una empresa tiene disponibles dos paredes como las siguientes que utilizan como murales para fijar carteles, afiches o pendones.

Pared 1



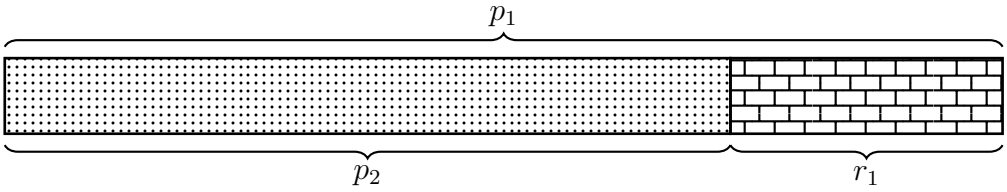
Pared 2



El gerente de la empresa desea fijar unos carteles del mismo ancho, con la condición de que tiene que colocar los afiches completos, que abarquen en su totalidad dichas paredes y que no se solapen. La empresa tiene la capacidad de mandar a imprimir y cortar los afiches de cualquier ancho. ¿Cuál será el afiche de mayor ancho que puede colocar la empresa de tal manera que utilice en su totalidad las paredes y que los afiches se peguen completos sin solaparse?

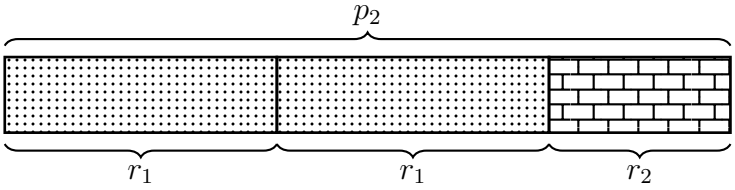
1. Para saber cuál es el afiche más ancho que se puede colocar en las paredes p_1 y p_2, se debe observar que la pared más corta es p_2, por lo tanto el afiche más ancho debe tener por mucho el ancho de esa pared.

2. Si a la pared más ancha p_1 se tapa con la pared más corta p_2, se obtiene un resto r_1 de pared como el siguiente



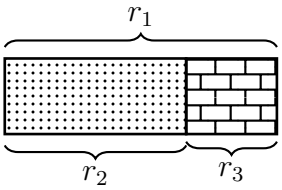
3. Como los afiches taparán completa y exactamente la pared p_2, para que estos afiches también tapen la pared p_1, entonces deben tapar completa y exactamente el resto r_1 de la pared. Por lo que para este caso el afiche más ancho debe tener por mucho el ancho de ese resto r_1 de pared.

4. El ancho de r_1 pasa a ser entonces el candidato a ser el ancho del afiche, por lo que es necesario que el afiche que tape la pared r_1 también tape la pared p_2. Así, si ahora se tapa pared p_2 con la pared r_1 tantas veces como sea posible, entonces, se obtiene un resto r_2 de pared como el siguiente

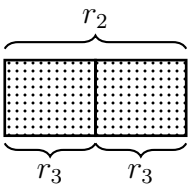


5. En este caso ocurre lo mismo que en el numeral 3, para tapar la pared p_2 se debe tapar también la pared restante r_2, por lo que el afiche más ancho debe tener por mucho el ancho de ese resto r_2 de pared.

6. De lo anterior, se tiene que el ancho de r_2 pasa a ser entonces el candidato a ser el ancho del afiche, por lo que es necesario que el afiche que tape la pared r_2 también tape la pared restante r_1. Así, si ahora se tapa pared r_1 con la pared r_2 tantas veces como sea posible, entonces, se obtiene un resto r_3 de pared como el siguiente

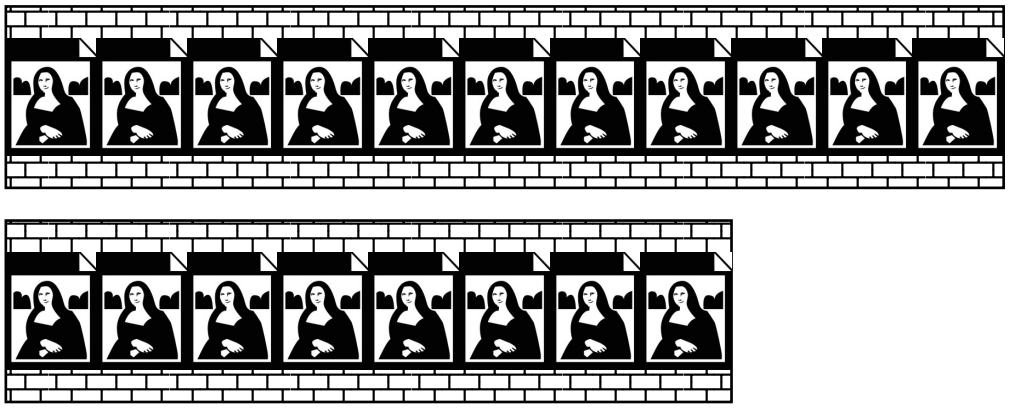


7. En este punto, el análisis es similar a los casos anteriores, pues para tapar la pared r_1 es necesario tapar el resto de pared r_3. Con lo cual se obtiene un nuevo candidato, el ancho de la pared r_3. Si con esta pared r_3 se tapa la pared r_2 tantas veces como sea posible, entonces se obtiene el siguiente cubrimiento total de la pared r_2.



8. Por la construcción anterior, se tiene que un afiche que utilice en su totalidad las paredes y que se peguen completos sin solaparse está dado por el ancho de la pared r_3. Un afiche de este ancho será el de mayor tamaño pues siempre se escogió el de mayor tamaño posible para ir descartando las otras opciones.

9. El aspecto de las paredes con los afiches colocados de acuerdo al resultado obtenido es el siguiente



No siempre este problema es solucionable, ya que existen paredes de distinta longitud, tales que no tienen un segmento que quepa una número exacto de veces, por ejemplo, si la primera longitud mide $l_1 = \sqrt{2}$ unidades y la segunda $l_2 = 2$, no existe un segmento que quepa un número exacto de veces, a este tipo de medidas se les denomina inconmensurables, y para las que sí existe un segmento que cabe un número exacto de veces se les llama conmensurables.

Volviendo al problema de encontrar el afiche de mayor longitud que quepa en un par de paredes de forma exacta sin solaparse, en el caso de que las paredes tengan longitudes números naturales unidades, entonces en éstas siempre es posible encontrar una longitud que cumpla con las condiciones impuestas anteriormente, pues estas longitudes con conmensurables, ya que en el peor de los casos los afiches con longitud una (1) unidad siempre cabra un número exacto de veces sin solaparse.

En matemáticas, a el segmento de mayor longitud en cabe un número exacto de veces en dos segmentos conmensurables se le conoce como el *máximo común divisor* de los dos segmentos.

A partir del análisis anterior, se puede diseñar una función recursiva que permite calcular el máximo común divisor de dos números p y q, donde se supone que $p \geq q$.

$$mcd\_recur(p, q) = m$$

donde se tienen las variables

$$\begin{aligned} p &:= \text{Primer número natural positivo} \\ q &:= \text{Segundo número natural positivo tal que } q \leq p \\ m &:= \text{Máximo común divisor de los números } p \text{ y } q \end{aligned}$$

entonces

$$\begin{aligned} mcd\_recur &: \mathbb{N} \times \mathbb{N} \to \mathbb{N} \\ (p, q) &\mapsto \begin{cases} p, & \text{si} \quad q = 0; \\ mcd\_recur(q, p \bmod q), & \text{en otro caso.} \end{cases} \end{aligned}$$

Como se desea que se pueda calcular el máximo común divisor de cualesquiera dos números naturales, entonces la función anterior se utilizará como una función auxiliar (*helper*), y la siguiente función sí permitirá calcular el máximo común divisor de cualesquiera dos números, ésta lo que hace es primero encontrar el mayor de los dos números y luego utilizar la función $mcd\_recur(p, q)$ de forma correcta.

$$mcd(p, q) = m$$

donde se tienen las variables

$$\begin{aligned} p &:= \text{Primer número natural positivo} \\ q &:= \text{Segundo número natural positivo} \\ m &:= \text{Máximo común divisor de los números } p \text{ y } q \end{aligned}$$

entonces

$$\begin{aligned} mcd &: \mathbb{N} \times \mathbb{N} \to \mathbb{N} \\ (p, q) &\mapsto \begin{cases} mcd\_recur(p, q), & \text{si} \quad p > q; \\ mcd\_recur(q, p), & \text{en otro caso.} \end{cases} \end{aligned}$$

Y la función
La codificación en C++ de estas funciones junto con su programa principal es

```
#include<iostream>
#include<cstdlib>

using namespace std;
```

```
int mcd_recur(int p, int q){
  if(q == 0){
    return p;
  }else{
    return mcd_recur(q,p % q);
  };
};
```

```
int mcd(int p, int q){
  if(p > q){
    return mcd_recur(p,q);
  }else{
    return mcd_recur(q,p);
  };
};
```

```
int main(){
  int p;
  int q;
  cout << "p? = ";
  cin >> p;
  cout << "q? = ";
  cin >> q;
  cout << "m.c.d(p,q) = ";
  cout << mcd(p,q);
  cout << "\n";
  system("pause");
  return EXIT_SUCCESS;
};
```

7.1. Teorema fundamental de la programación recursiva

Teorema (Teorema fundamental de la programación recursiva)**.** *Un lenguaje de programación es completo en Turing si tiene valores enteros no negativos, funciones aritméticas elementales sobre dichos valores, así como un mecanismo para definir nuevas funciones utilizando las funciones ya existentes, la selección (*if*) y la recursión.*

7.2. Ejercicios

1. Modele mediante una función matemática y diseñe un programa recursivo que determine el mayor de dos números enteros no negativos que utilice sólo el operador de comparación de la igualdad (`==`), la función sucesor (sumar `1`), la función predecesor (restar `1`) y la estructura condicional (`if`, `if-else`).

2. Modele mediante una función matemática y diseñe un programa recursivo que calcule la suma de los primeros n números positivos $\left(\sum_{i=1}^{n} i\right)$.

3. Modele mediante una función matemática y diseñe un programa recursivo que calcule la suma de los cuadrados de los primeros n números positivos $\left(\sum_{i=1}^{n} i^2\right)$.

4. Modele mediante una función matemática y diseñe un programa recursivo que calcule el producto de los cuadrados de los primeros n números positivos $\left(\prod_{i=1}^{n} i^2\right)$.

5. Modele mediante una función matemática y diseñe un programa recursivo que calcule el logaritmo entero en base 2 de n $\left(\left\lfloor \log_2 n \right\rfloor\right)$. Por ejemplo, $\left\lfloor \log_2 1 \right\rfloor = 0$, $\left\lfloor \log_2 4 \right\rfloor = 2$, $\left\lfloor \log_2 7 \right\rfloor = 2$, $\left\lfloor \log_2 15 \right\rfloor = 3$.

6. Modele mediante una función matemática y diseñe un programa recursivo que calcule el logaritmo entero en base b de n $\left(\left\lfloor \log_b n \right\rfloor\right)$.

7. Modele mediante una función matemática y diseñe un programa recursivo que calcule la raíz cuadrada entera de a $\left(\left\lfloor \sqrt{a} \right\rfloor\right)$. Por ejemplo, $\left\lfloor \sqrt{0} \right\rfloor = 0$, $\left\lfloor \sqrt{1} \right\rfloor = 1$, $\left\lfloor \sqrt{5} \right\rfloor = 2$, $\left\lfloor \sqrt{10} \right\rfloor = 3$.

8. Modele mediante una función matemática y diseñe un programa recursivo que calcule la raíz n-ésima entera de a $\left(\left\lfloor \sqrt[n]{a} \right\rfloor\right)$.

9. Modele mediante una función matemática y diseñe un programa recursivo que calcule la función módulo (m mód $n = k$). Por ejemplo, 0 mód 2 = 0, 4 mód 2 = 0, 3 mód 3 = 0, 10 mód 3 = 1, 14 mód 5 = 4.

10. Modele mediante una función matemática y diseñe un programa recursivo que determine la cantidad de dígitos que componen un número natural n. Por ejemplo, $longitud(654321) = 6$.

11. Modele mediante una función matemática y diseñe un programa recursivo que invierta la cifras de un número n dado. Por ejemplo, $inversa(654321) = 123456$.

12. Modele mediante una función matemática y diseñe un programa recursivo que determine si un número es palíndromo. Un número se dice palíndromo si al leerlo de izquierda a derecha es lo mismo que leerlo de derecha a izquierda. Por ejemplo, $palindromo(1) = V$, $palindromo(1234321) = V$, $palindromo(123421) = F$.

13. Modele mediante una función matemática y diseñe un programa recursivo que calcule el *mínimo común múltiplo* de dos números positivos a y b. Por ejemplo, $mcm(18, 24) = 72$.

14. Modele mediante una función matemática y diseñe un programa que dados dos número positivos p y q, donde p representa el numerador y q el denominador de la fracción $\frac{p}{q}$, imprima primero el numerador de la fracción $\frac{p}{q}$ simplificada a su mínima expresión y luego el denominador de la fracción $\frac{p}{q}$ simplificada a su mínima expresión.

15. Modele mediante una función matemática y diseñe un programa que dados cuatro número positivos p, q, r y s, donde p representa el numerador y q el denominador de la fracción $\frac{p}{q}$, y r representa el numerador y s el denominador de la fracción $\frac{r}{s}$, imprima primero el numerador de la fracción resultante de la operación $\frac{p}{q} + \frac{r}{s}$ simplificada a su mínima expresión y luego el denominador de la fracción resultante de la operación $\frac{p}{q} + \frac{r}{s}$ simplificada a su mínima expresión.

Capítulo 8

Estructuras de programación cíclicas

8.1. La estructura de control de ciclos *mientras* (`while`)

El ciclo `while` permite ejecutar un bloque de instrucciones mientras que una expresión booleana dada se cumpla, es decir, mientras su evaluación dé como resultado verdadero. La expresión booleana se denomina condición de parada y siempre se evalúa antes de ejecutar el bloque de instrucciones. Si la condición no se cumple, el bloque no se ejecuta. Si la condición se cumple, el bloque se ejecuta, después de lo cual la instrucción vuelve a empezar, es decir, la condición se vuelve a evaluar.

En el caso en que la condición se evalúe la primera vez como falsa, el bloque de instrucciones no será ejecutado, lo cual quiere decir que el número de repeticiones o iteraciones de este bloque será cero. Si la condición siempre evalúa a verdadero, la instrucción se ejecutará indefinidamente, es decir, un número infinito de veces.

La sintaxis general de un ciclo `while` es la siguiente

```
<init>
while(<cond>){
  <body>
  <update>
};
```

en donde:

- El fragmento `<init>` es el bloque de instrucciones donde se inicializan las variables que intervienen en la condición de parada.
- El fragmento `<cond>` es la condición de parada que se evalúa cada vez que se inicia el ciclo.
- El fragmento `<body>` es el bloque de instrucciones principal del ciclo que se ejecuta mientras la condición se cumpla.
- El fragmento `<update>` es el bloque que se utiliza para actualizar las variables que son utilizadas para evaluar la condición de parada cuando se intenta reiniciar el ciclo.

Ejemplo. Para el siguiente fragmento de código que contiene un ciclo `while`

```
int i = 0;
while(i <= 5){
  cout << i;
  cout << "\n";
  i = i + 1;
};
```

se tiene que el fragmento de código:

- `<init>` corresponde a la instrucción
  `int i = 0;`
- `<cond>` corresponde a la instrucción
  `i <= 5`
- `<body>` corresponde a las instrucciones
  `cout << i;`
  `cout << "\n";`
- `<update>` corresponde a la instrucción
  `i = i + 1;`

cuando se ejecuta este ciclo lo que se obtiene en la consola de salida es el texto que se presenta en el cuadro a la derecha

```
int i = 0;
while(i <= 5){
  cout << i;
  cout << "\n";
  i = i + 1;
};
```

```
0
1
2
3
4
5
```

en este caso la salida que se produce es la anterior porque el bloque

```
cout << i;
cout << "\n";
```

se ejecuta seis veces, variando `i` desde `0` hasta cuando `i` toma el valor `6`, que hace que la condición se evalué falso; obsérvese que la variable termina el ciclo con valor `i` = `6`, pero este valor no se imprime pues para este caso la condición se evalúa falso.

Ejemplo. Para el siguiente fragmento de código que contiene un ciclo `while`

```
int i = 1;
int j = 10;
while(i < j){
  cout << i;
  cout << " ";
  cout << j;
  cout << "\n";
  i = i * 2;
  j = j + 10;
};
```

la variables `i` y `j` se inicializan con los valores `1` y `10` respectivamente, luego se verifica que `1` sea menor estrictamente que `10`, a continuación se imprime el valor de la variable `i` seguido por un espacio, seguido por el valor de la variable `j`, seguido de una salto de línea; a continuación se multiplica la variable `i` por `2` y a la variable `j` se le suma `10`. Esto se realiza hasta que el valor de la variable `i` sea mayor o igual a el valor de la variable `j`.

El resultado de la ejecución de este ciclo mostrado en la consola de salida es el texto que se presenta en el cuadro a la derecha.

```
int i = 1;
int j = 10;
while(i < j){
  cout << i;
  cout << " ";
  cout << j;
  cout << "\n";
  i = i * 2;
  j = j + 10;
};
```

```
1 10
2 20
4 30
8 40
16 50
32 60
64 70
```

Obsérvese que las variables `i` y `j` terminan el ciclo con los valores `128` y `80`, y como `128` no es menor que `80`, entonces el ciclo se para, por esta razón no se imprimen estos valores.

Ejemplo. *El mínimo número positivo de la máquina*

Dado que los números reales que son representables en un computador son finitos, entonces es posible hablar del menor número positivo representable en la máquina, es decir el número

$$x_{\text{mín}} = \text{mín}\left\{x : (x \text{ es un número de máquina}) \wedge (x > 0)\right\}$$

Para encontrar dicho número hay un algoritmo muy sencillo que permite encontrar el valor.

El algoritmo consiste en calcular los términos de una progresión geométrica que inicia con el término $x_0 = 1$ y para la cual los términos siguientes se calculan utilizando la razón de la progresión $r = 1/2$, es decir, $x_{n+1} = \dfrac{x_n}{2}$, esto se realiza mientras cada nuevo término es positivo.

La codificación en C++ de una función que permite hallar el mínimo número positivo representable en la máquina junto con su programa principal es

```
#include<iostream>
#include<cstdlib>

using namespace std;
```

```
double min_maquina(){
  double Xo = 1.0;
  double Xi = Xo / 2;
  while(Xi > 0.0){
    Xo = Xi;
    Xi = Xo / 2.0;
  };
  return Xo;
};
```

como para calcular cada término de la progresión geométrica se necesita únicamente el término anterior, entonces son necesarias sólo dos variables, las cuales se utilizarán de la siguiente manera:

- La variable `Xo` representa el término x_n y se inicializa con el valor $x_0 =$ `1`.
- La variable `Xi` representa el término x_{n+1} y se inicializa con el valor `Xo / 2`.
- Ahora, dentro del ciclo la variable `Xo` jugará el rol del término x_{n+1} mediante la asignación `Xo = Xi;`.
- A la variable `Xi` se le asigna el siguiente término de la progresión calculado y asignado mediante la expresión `Xi = Xo / 2.0;`.
- Las dos últimas rutinas descritas se realizan mientras el valor de la variable `Xi` sea mayor a `0`.
- El ciclo en algún momento deja de ejecutarse ya que el conjunto de los números de máquina es finito, la progresión geométrica es decreciente y está acotada inferiormente por `0`.
- Finalmente, se retorna el valor almacenado en la variable `Xo` pues fue el último término de la progresión que era distinto de `0`.

```
int main(){
  cout << "El numero positivo mas pequeno ";
  cout << "que se puede representar en la maquina es: ";
  cout << min_maquina();
  cout << "\n";
  system("pause");
  return EXIT_SUCCESS;
};
```

Si se ejecuta el anterior programa el resultado que se obtiene es el siguiente

```
El numero positivo mas pequeno que se puede
representar en la maquina es: 4.94066e-324
Presione una tecla para continuar . . .
```

8.2. La estructura de control de ciclos *para* (for)

Existe otra estructura cíclica que es de uso frecuente en programación, ésta se conoce como un ciclo `for`. Esta estructura es equivalente a la estructura `while`, pero tiene la ventaja de que es más compacta y es usualmente utilizada cuando se conocen los valores inicial y final de la variable que es utilizada en la condición de parada.

Dada la sintaxis general de un ciclo `while`

```
<init>
while(<cond>){
  <body>
  <update>
};
```

la sintaxis general de un ciclo `for` que es equivalente al ciclo `while` es

```
for(<init> ; <cond> ; <update>){
  <body>
};
```

Ejemplo. *La suma de los primeros n números naturales*

Las dos siguientes funciones permiten calcular la suma de los primeros n números naturales positivos, es decir, permiten calcular el valor de la expresión

$$1+2+3+\cdots+(n-1)+n \quad \text{que abreviadamante se escribe como} \quad \sum_{i=1}^{n} i$$

```
int suma(int n){
  int s = 0;
  int i = 1;
  while(i <= n){
    s = s + i;
    i++;
  };
  return s;
};
```

```
int suma(int n){
  int s = 0;
  for(int i = 1 ; i <= n ; i++){
    s = s + i;
  };
  return s;
};
```

estas dos funciones son equivalentes, ya que ejecutan las mismas modificaciones de las variables, pues se tiene que el fragmento de código:

- `<init>` corresponde a la instrucción
  `int i = 1;`
- `<cond>` corresponde a la instrucción
  `i <= n`
- `<body>` corresponde a la instrucción
  `s = s + i;`
- `<update>` corresponde a la instrucción
  `i++;`

En la construcción de estas funciones aparecen dos variable que tienen una connotación muy importante:

- La variable `i` juega el rol de una *variable contadora* ya que permite llevar el conteo de cuantos ciclos se han efectuado.
- La variable `s` juega el rol de *variable acumuladora* pues en ésta se acumula o almacena el valor parcial que se desea calcular utilizando el ciclo.

La codificación en C++ de una función que permite sumar los primeros n números naturales positivos junto con su programa principal es

```
#include<iostream>
#include<cstdlib>

using namespace std;
```

```
int suma(int n){
  int s = 0;
  for(int i = 1 ; i <= n ; i++){
    s = s + i;
  };
  return s;
};
```

```
int main(){
  int n;
  cout << "n? = ";
  cin >> n;
  cout << "La suma de los primeros n numeros es: ";
  cout << suma(n);
  cout << "\n";
  system("pause");
  return EXIT_SUCCESS;
};
```

Si se ejecuta el anterior programa y como entrada se ingresa el valor `n = 6` (por ejemplo el número de caras de un dado ⚀ + ⚁ + ⚂ + ⚃ + ⚄ + ⚅), el resultado que se obtiene es el siguiente

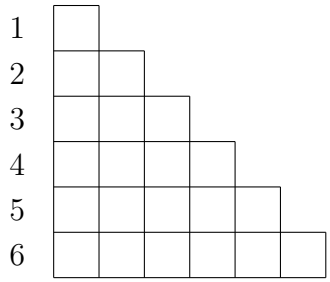
```
n? = 6
La suma de los primeros n numeros es: 21
Presione una tecla para continuar . . .
```

En general es fácil comprobar si el resultado que se obtiene utilizando la función `suma(n)` es correcto, pues existe una fórmula muy sencilla para calcular la suma de los primeros n números naturales positivos sin necesidad de realizar la suma exhaustivamente. Esta fórmula es

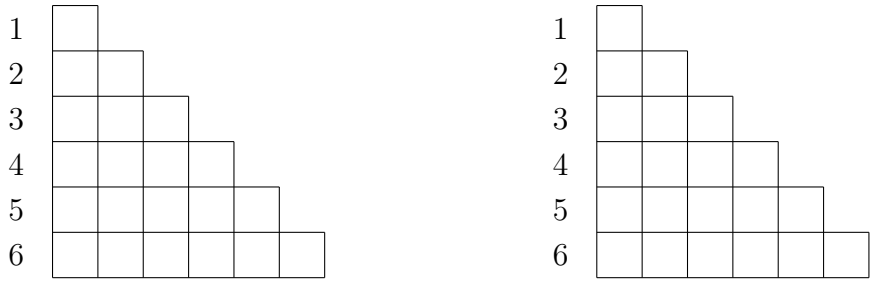
$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

la cual se puede demostrar por inducción matemática o se puede generalizar a partir de la observación de las siguientes figuras

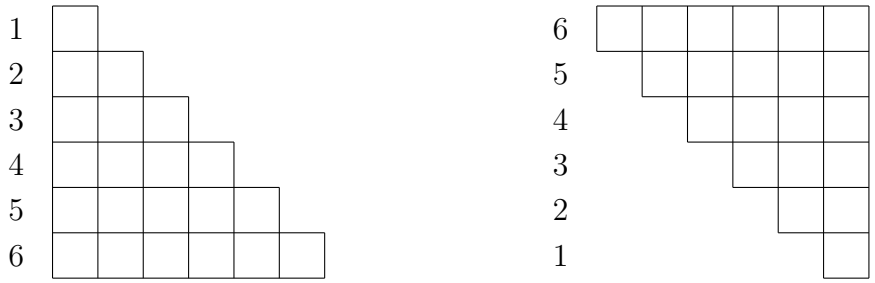
Al sumar los números del 1 al 6 se puede construir una escalera de cuadrados descendente empezando por un cuadrado y terminando con seis cuadrados, como la que se muestra a continuación,



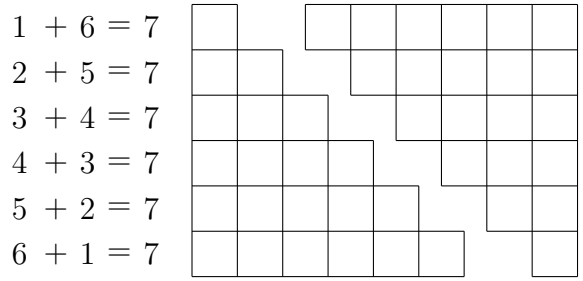
luego, si se duplica la escalera se obtienen las siguientes figuras



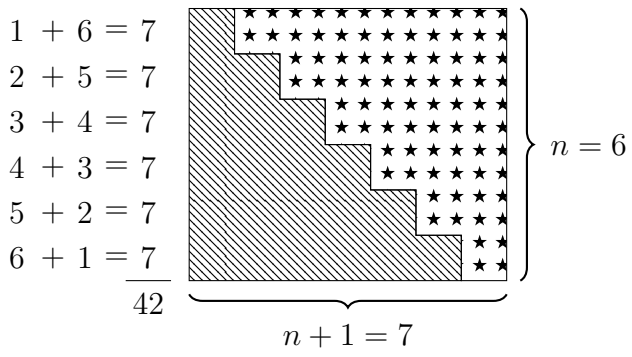
ahora, si se rota la segunda escalera dos ángulos rectos se obtienen las siguientes figuras



si se suman por filas la cantidad de cuadrados se obtiene siempre el mismo resultado



como se observa a continuación, siempre se puede formar un rectángulo tal que uno de sus lados en igual a $n = 6$ y el otro igual a $n + 1 = 7$, y por la forma en que se construyó el rectángulo se observa que la suma de los cuadrados de cada escalera es igual a la mitad de la cantidad total de cuadrados en el rectángulo.



es decir, que para el valor $n = 6$ se obtiene el resultado

$$1+2+3+4+5+6 = \sum_{i=1}^{6} i = \frac{6(6+1)}{2} = \frac{6*7}{2} = \frac{42}{2} = 21$$

el cual es el mismo valor que se obtuvo con el uso de la función `suma(n)`.

8.3. La estructura de control de ciclos *hacer-mientras* (do)

Existe otra estructura cíclica en programación, ésta se conoce como un ciclo `do`. Esta estructura es equivalente a la estructura `while`, es usualmente utilizada cuando se sabe con anticipación que se hará al menos una evaluación del bloque principal del ciclo. En esta estructura cíclica la verificación de la condición de parada se realiza al final del ciclo.

Dada la sintaxis general de un ciclo `while`

```
<init>
while(<cond>){
  <body>
  <update>
};
```

la sintaxis general de un ciclo `do` que es equivalente al ciclo `while` es

```
<init>
do{
  <body>
  <update>
}while(<cond>);
```

Ejemplo. *El mínimo número positivo de la máquina (versión `do`)*

Como para el cálculo del mínimo número positivo de la máquina se utilizó un ciclo mientras y cuando se hace uso del ciclo siempre se ejecutan las instrucciones dentro del ciclo al menos una vez, es posible utilizar un ciclo `do` en una nueva función equivalente a la que tiene el ciclo `while` así como se muestra a continuación

```
double min_maquina(){
  double Xo = 1.0;
  double Xi = Xo / 2;
  while(Xi > 0.0){
    Xo = Xi;
    Xi = Xo / 2.0;
  };
  return Xo;
};
```

⇔

```
double min_maquina(){
  double Xo = 1.0;
  double Xi = Xo / 2;
  do{
    Xo = Xi;
    Xi = Xo / 2.0;
  }while(Xi > 0.0);
  return Xo;
};
```

Ejemplo. *Raíz cuadrada de un número real positivo*

Calcular una raíz cuadrada es una labor habitual, hasta la más sencilla calculadora tiene la capacidad de calcular raíces de números positivos.

En primera instancia se justificará por que siempre se puede hablar de la raiz cuadrada de un número positivo, ya que esta siempre existe. En segunda instancia se explicará un método para aproximar la raíz cuadrada tanto como sea necesario. En tercera instancia se explicará el criterio de parada y finalmente se mostrará el código implementado en C++.

Dado $a \in \mathbb{R}^+$ siempre existe un valor $d \in \mathbb{R}^+$ llamado la raiz cuadrada de a tal que $d^2 = a$. Esto se tiene ya que la función $f(x) = x^2 - a$ es continua, $f(0) = -a < 0$, $f(1+a) = 1+a+a^2 > 0$, entonces por el teorema del valor intermedio existe $d \in (0, 1+a)$ tal que $f(d) = 0 = d^2 - a$, es decir, el polinomio $f(x)$ tiene una raiz positiva, de lo cual se concluye que $d^2 = a$.

Ahora que se demostró que la raiz cuadrada existe para todo número real positivo a, entonces se va a introducir un método constructivo para encontrar una aproximación a dicho valor.

- Como primer paso en el método se da un valor inicial x_0 que es cualquier número real positivo que servirá como una primera aproximación a la raiz cuadrada que se desea calcular, $x_0 \approx \sqrt{a}$.

- En segunda instancia, por la ley de tricotomía de los números reales, se cumple que:

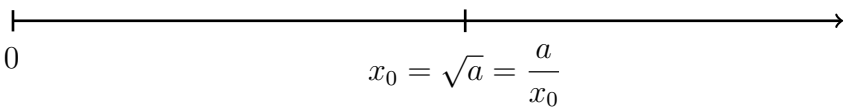
$$x_0 = \sqrt{a} \qquad \vee \qquad x_0 > \sqrt{a} \qquad \vee \qquad x_0 < \sqrt{a}$$

  a partir de estos casos se pueden hacer los siguientes análisis:

- Si $x_0 = \sqrt{a}$ entonces

$$\begin{aligned} x_0 &= \sqrt{a} \\ x_0^2 &= a \\ x_0 &= \frac{a}{x_0} \end{aligned}$$

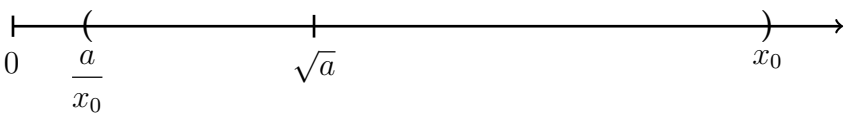
de donde $x_0 = \sqrt{a} = \dfrac{a}{x_0}$.



- Si $x_0 > \sqrt{a}$ entonces

$$\begin{aligned} x_0 &> \sqrt{a} \\ x_0\sqrt{a} &> \sqrt{a}\sqrt{a} \\ x_0\sqrt{a} &> a \\ \sqrt{a} &> \frac{a}{x_0} \end{aligned}$$

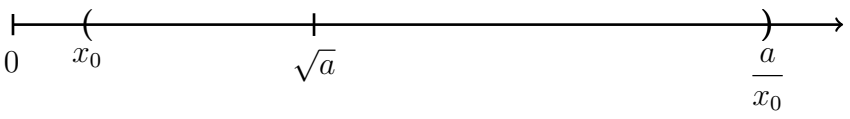
de donde $x_0 > \sqrt{a} > \dfrac{a}{x_0}$.



- Si $x_0 < \sqrt{a}$ entonces

$$\begin{aligned} x_0 &< \sqrt{a} \\ x_0\sqrt{a} &< \sqrt{a}\sqrt{a} \\ x_0\sqrt{a} &< a \\ \sqrt{a} &< \frac{a}{x_0} \end{aligned}$$

de donde $x_0 < \sqrt{a} < \dfrac{a}{x_0}$.



- Como se observa en el análisis anterior, la relación de $\sqrt{a}$ con x_0 y $\dfrac{a}{x_0}$, es que $\sqrt{a}$ siempre o es igual a esos valores (primer caso) o está entre esos valores (segundo y tercer caso).

- Si $\sqrt{a}$ es igual a x_0 y $\dfrac{a}{x_0}$, entonces el problema ya se soluciono y se tiene el valor exacto de $\sqrt{a}$.

- Si $\sqrt{a}$ está entre x_0 y $\dfrac{a}{x_0}$, entonces hay que decidir si alguno de esos valores es una aproximación lo suficientemente exacta a $\sqrt{a}$, o si una combinación esos valores es una aproximación lo suficientemente exacta a $\sqrt{a}$, o si es necesario calcular valores que se aproximen mejor al valor exacto de $\sqrt{a}$.

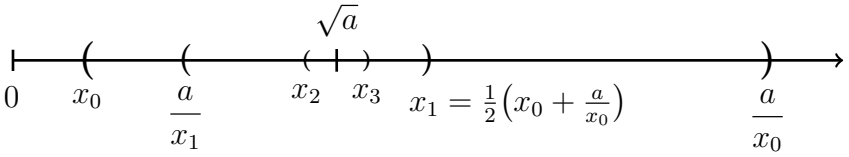
- Si es necesario calcular valores que se aproximen mejor al valor exacto de $\sqrt{a}$, puede observarse que como $\sqrt{a}$ está entre x_0 y $\dfrac{a}{x_0}$, entonces el punto medio del intervalo que determinan esos valores $\frac{1}{2}\left(x_0 + \frac{a}{x_0}\right)$ es un valor que también sirve de aproximación al valor de $\sqrt{a}$, por lo tanto ese valor se puede tomar como una nueva aproximación inicial al valor de $\sqrt{a}$.

- Si se sigue el procedimiento de calcular el valor medio del intervalo que se construye a partir de cada nueva aproximación, entonces lo que se obtiene es una sucesión de valores $x_0, x_1, x_2, \ldots, x_i, \ldots$ que cada vez se aproximan mejor (que convergen) al valor de $\sqrt{a}$.

- La sucesión que se obtiene se puede expresar en general de la siguiente manera:

$$x_0 := \text{cualquier valor real positivo fijo}$$
$$x_{i+1} := \frac{1}{2}\Big(x_i + \frac{a}{x_i}\Big)$$

obsérvese que cada vez que se calcula un nuevo valor, estos valores se encuentran alternados alrededor de $\sqrt{a}$ y aproximándose cada vez mejor.



- Una pregunta que hay que hacerse en este momento es ¿hasta cuándo se deben calcular nuevos valores?.

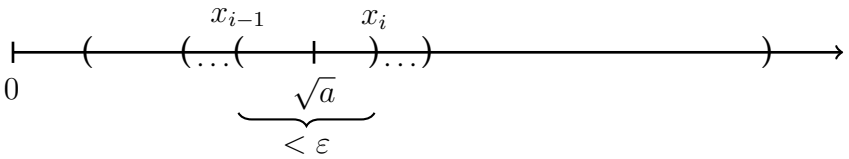
  Una de las características de estos métodos constructivistas es que es posible que no se obtenga el valor exacto porque el resultado no se puede representar con finita memoria o es demasiado grande para la memoria o el tiempo necesario para calcular el valor exacto es infinito.

  Para este ejemplo y muchos de este tipo, hay que conformarse con obtener una aproximación lo suficientemente precisa que sea útil para resolver el problema que se ésta solucionado.

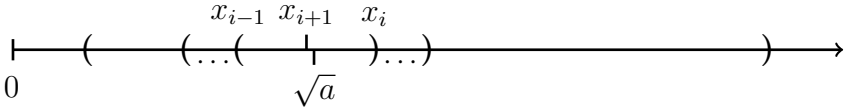
  Para decidir cuando se tiene una aproximación lo suficientemente precisa de la solución, el método más sencillo que se suele utilizar es detenerse cuando la distancia entre dos aproximaciones seguidas es lo suficientemente pequeña con respecto a un valor dado, lo que se suele denominar la *precisión del error* y se denota por el símbolo ε. A la distancia entre las dos aproximaciones se les llama el *error absoluto* y normalmente se calculan nuevas aproximaciones hasta que el error absoluto sea menor que ε, lo que se expresa como

$$|x_i - x_{i-1}| < \varepsilon$$

  y que gráficamente se puede representar así



- Cuando se detiene el calculo de nuevas aproximaciones, entonces el resultado que se suele retornar es el punto medio de los dos últimos valores calculados



La codificación en C++ de una función que permite calcular la raíz cuadrada de cualquier número real positivo junto con su programa principal es

```
#include<iostream>
#include<cstdlib>

using namespace std;
```

La función valor absoluto es necesaria para calcular la distancia entre dos valores.

```
double valor_absoluto(double x){
  if(x >= 0){
    return x;
  }else{
    return -x;
  };
};
```

Como para calcular el error relativo es necesario tener al menos dos aproximaciones, entonces, dado el primer valor inicial es necesario calcular al menos otro valor de la sucesión, razón por la cual el uso de un ciclo `do` es muy conveniente.

En esta función como valor inicial se va a tomar el valor al cual se le va a hallar la raíz cuadrada (a), pero no esto necesario, no olvide que el método funciona para cualquier número real positivo.

Adicionalmente, en la notación de la función la variable `Xo` se utiliza para almacenar la aproximación anterior a la raíz y la variable `Xi` se utiliza para almacenar la aproximación siguiente a la raíz.

La precisión del error que se utiliza aquí es $\varepsilon = 10^{-4}$, pero también se podría pasar como un nuevo parámetro de la función.

```
double raiz(double a){
  double Xo;
  double Xi = a;   // valor inicial
  do{
    Xo = Xi;
    Xi = 0.5 * (Xo + a / Xo);
  }while(valor_absoluto(Xo - Xi) >= 1e-4);
  return 0.5 * (Xi + a / Xi);
};
```

```
int main(){
  double a;
  cout << "a? = ";
  cin >> a;
  cout << "Una aproximacion de la raiz cuadrada de a es: ";
  cout << raiz(a);
  cout << "\n";
  system("pause");
  return EXIT_SUCCESS;
};
```

Si se ejecuta el anterior programa y como entrada se ingresa el valor `a=2`, el resultado que se obtiene es el siguiente

```
a? = 2
Una aproximacion de la raiz cuadrada de a es: 1.41421
Presione una tecla para continuar . . .
```

El valor 1.41421 es una aproximación al valor $\sqrt{2}$ con una precisión del error menor a 10^{-4}.

El método anterior es equivalente al método de bisección y el de Newton para hallar raíces de funciones.

8.4. Simulación de ciclos usando funciones recursivas

Como todo ciclo `for` y `do` puede ser simulado en términos de ciclos `while`, y estos a su vez se podrán simular mediante funciones recursivas, como se explicará más adelante, entonces todo ciclo puede ser simulado mediante funciones recursivas.

Dado un bloque general de un ciclo `while`

```
<init>
while(<cond>){
  <body>
  <update>
};
```

este bloque se puede simular mediante funciones recursivas de la siguiente manera:

1. Se debe crear una nueva función `recHelperFunc()` que tendrá tantos parámetros como variables intervengan en el ciclo `while` a simular;

2. La variable principal que se desea calcular con el ciclo se debe colocar como primer parámetro de la función.

3. En la función se debe agregar una estructura condicional tal que su condición es la misma que la del ciclo `while`.

4. Dentro del condicional se anexan las instrucciones del `<body>` y luego las del `<update>`.

5. Finalmente se agrega una asignación del resultado de llamar recursivamente la función `recHelperFunc()` a la variable que se desea calcular con el ciclo. La función se llama con los mismo parámetros iniciales, pero que fueron modificados dentro de las instrucciones previas al nuevo llamado recursivo de la función dentro del condicional.

6. Finalmente se retorna la variable principal que se desea calcular con el ciclo inmediatamente después de finalizar el condicional.

```
int recHelperFunc(int var1, int var2, ... , int varN) {
  if(<cond>) {
    <body>
    <update>
    var1 = recHelperFunc(var1, var2, ... , varN); // llamada recursiva
  };
  return var1;
};
```

7. La porción del código que abarca el ciclo `while` se reemplaza por una asignación donde a la variable principal que se desea calcular con el ciclo se le asigna el resultado de evaluar la función recursiva en las variables que intervienen en el ciclo y que fueron inicializadas previamente al ciclo.

```
<init>
while(<cond>){
  <body>
  <update>
};
```

⇕

```
<init>
var1 = recHelperFunc(var1, var2, ... , varN);
```

Ejemplo. *El factorial de un número con ciclos y con funciones recursivas*

Para la siguiente función que permite calcular el factorial de un número y que utiliza un ciclo `while` para el cálculo del factorial, se construirá una nueva función que simule el ciclo `while` usando una función recursiva utilizando el método descrito anteriormente y que es diferente a la presentada en el capítulo de funciones recursivas.

```
int factorial(int n){
  int fact = 1;
  int i = 2;
  while(i <= n){
    fact *= i;
    i++;
  };
  return fact;
};
```

La función recursiva auxiliar que se va a construir es la función `recHelperFuncFact()` la cual tendrá 3 parámetros, que son las variables que intervienen en el ciclo `while` de la función original. El ciclo está diseñado para calcular el valor de la variable `fact`, por lo tanto ésta será la primera variable que se le pasé a la nueva función recursiva y será la que retorne la función recursiva cada vez que se ejecute, el aspecto de esta función es el siguiente.

```
int recHelperFuncFact(int fact, int i, int n) {
  if(i <= n) {
    fact *= i;
    i++;
    fact = recHelperFuncFact(fact, i, n); // llamada recursiva
  };
  return fact;
};
```

La función original será modificada eliminando el ciclo `while` sin eliminar el bloque `<init>` y reemplazándola por la instrucción donde se llama la función recursiva con las variables que intervienen en el ciclo original y se asigna el valor final de la evaluación de la función recursiva a la variable `fact` que se deseaba calcular en el ciclo. El aspecto de la función original modificada como se explicó anteriormente es el siguiente.

```
int factorial(int n){
  int fact = 1;
  int i = 2;
  fact = recHelperFuncFact(fact, i, n);
  return fact;
};
```

8.5. Teorema fundamental de la programación estructurada

Teorema (Teorema fundamental de la programación estructurada)**.** *Un lenguaje de programación es completo en Turing siempre que tenga variables enteras no negativas y las operaciones aritméticas elementales sobre dichas variables, y que ejecute enunciados en forma secuencial, incluyendo enunciados de asignación (=), selección (*if*), y ciclos (*while*).*

8.6. Validación de datos usando ciclos

Con respecto a la validación hecha en el capítulo donde se explicó la estructura condicional, en la cual si había algún error al introducir los datos, entonces el programa terminaba sin que hubiese alguna otra opción adicional a la de informar al sistema operativo que se salio con una falla generada durante la ejecución del programa.

Otra forma de realizar una validación es el uso del ciclo `while`. Para codificar la validación con el ciclo `while` se leen los valores de las variables que deben ser introducidas por el usuario mediante el teclado, luego se verifica si se incumple con alguna de las condiciones que deben cumplir las variables, en caso de que se incumpla, se le debe pedir al usuario que vuelva a ingresar la información y esto se hace hasta que la información ingresada sea correcta.

Ejemplo. Con respecto al algoritmo para el cálculo del área de un rectángulo dada la definición del tipo de dato podrían estarse leyendo largos o anchos negativos. La notación matemática también permite restringir el dominio y el rango de los conjuntos quedando la función de la siguiente forma

$$\begin{aligned} area\_rectangulo : \mathbb{R}^{0,+} \times \mathbb{R}^{0,+} &\to \mathbb{R}^{0,+} \\ (l, a) &\mapsto l * a \end{aligned}$$

Un programa que permite calcular el área de un rectángulo y que hace la validación de las variables ingresadas mediante el uso de ciclos y que no se termina si el usuario ingresa un valor errado es

La codificación en C++ de la función para calcular el área de un rectángulo haciendo la validación tanto del ancho como de la altura usando ciclos es la siguiente

```
#include<iostream>
#include<cstdlib>

using namespace std;
```

```
double area_rectangulo(double l, double a){
  return l*a;
};
```

Obsérvese que la validación se realiza en función principal y no en la función para calcular el área del rectángulo, durante la validación se verifica si alguna de las longitudes es negativa, en caso de que esto ocurra, se le informa al usuario que la dimensión no es valida y se le pide que vuelva a ingresar la dimensión.

```
int main(){
  double largo;
  double ancho;
  cout << "largo? = ";
  cin >> largo;
  while(largo < 0){
    cout << "El largo no es valido";
    cout << "\n";
    cout << "largo? = ";
    cin >> largo;
  };
  cout << "ancho? = ";
  cin >> ancho;
  while(ancho < 0){
    cout << "El ancho no es valido";
    cout << "\n";
    cout << "ancho? = ";
    cin >> ancho;
  };
  cout << "area rectangulo: ";
  cout << area_rectangulo(largo, ancho);
  cout << "\n";
  system("pause");
  return EXIT_SUCCESS;
};
```

8.7. Ejercicios

1. Imprimir un listado con los números del 1 al 100 cada uno con su respectivo cuadrado.

2. Imprimir un listado con los números impares desde 1 hasta 999 y seguidamente otro listado con los números pares desde 2 hasta 1000.

3. Imprimir los números pares en forma descendente hasta 2 que son menores o iguales a un número natural $n \geq 2$ dado.

4. Imprimir los 50 primeros números de Fibonacci. Recuerde que un número de Fibonacci se calcula como la suma de los dos anteriores así: $0, 1, 1, 2, 3, 5, 8, 13$.

5. Imprimir los números de 1 hasta un número natural n dado, cada uno con su respectivo factorial.

6. Calcular el valor de 2 elevado a la potencia n.

7. Leer un número natural n, leer otro dato de tipo real x y calcular x^n.

8. En 2010 el país A tiene una población de 25 millones de habitantes y el país B de 19.9 millones. Las tasas de crecimiento anual de la población son de $2\,\%$ y $3\,\%$ respectivamente. Desarrollar un algoritmo para informar en que año la población del país B supera a la de A.

9. Elaborar una función que reciba un número entero, que retorne -1 si el número es negativo, si el número es positivo debe devolver una clave calculada de la siguiente manera: se suma cada dígito que compone el número y a esa suma se le calcula el módulo 7. Por ejemplo: para la cifra 513, la clave será $5 + 1 + 3 = 9$; $9 \mod 7 = 2$.

10. Calcule un programa que muestre las tablas de multiplicar del 1 al 9.

11. Introducir un rango especificado por 2 números enteros, tal que el primero sea menor al segundo y contar el número de múltiplos de un numero entero leído que existe en el rango. Por ejemplo, si se introduce 2 y 21, el número de múltiplos de 3 es 7, dado que $3, 6, 9, 12, 15, 18$ y 21 son múltiplos de 3 en el rango $[2, 21]$.

12. Diseñar una función que permita calcular una aproximación de la función exponencial alrededor de 0 para cualquier valor $x \in \mathbb{R}$, utilizando los primeros n términos de la serie de Maclaurin
$$\exp(x) \approx \sum_{i=0}^{n} \frac{x^i}{i!}.$$

13. Diseñar una función que permita calcular una aproximación de la función seno alrededor de 0 para cualquier valor $x \in \mathbb{R}$ (x dado en radianes), utilizando los primeros n términos de la serie de Maclaurin
$$\operatorname{sen}(x) \approx \sum_{i=0}^{n} \frac{(-1)^i x^{2i+1}}{(2i+1)!}.$$

14. Diseñar una función que permita calcular una aproximación de la función coseno alrededor de 0 para cualquier valor $x \in \mathbb{R}$ (x dado en radianes), utilizando los primeros n términos de la serie de Maclaurin

$$\cos(x) \approx \sum_{i=0}^{n} \frac{(-1)^i x^{2i}}{(2i)!}.$$

15. Diseñar una función que permita calcular una aproximación de la función arco tangente para cualquier valor $x \in [-1, 1]$, utilizando los primeros n términos de la serie de Maclaurin (al evaluar esta función el resultado que se obtiene esta expresado en radianes)

$$\arctan(x) \approx \sum_{i=0}^{n} \frac{(-1)^i x^{2i+1}}{(2i+1)}.$$

Capítulo 9

Flujos de entrada y salida

9.1. Definición

Un *flujo* es un objeto desde el cual se puede enviar o recibir información.

- Para el caso de una fuente de entrada de información, ésta puede ser enviada desde el teclado o desde un archivo o desde una red local de comunicaciones o desde un nodo de internet, entre otros.
- Para el caso de una fuente de salida de información, ésta típicamente es enviada a la consola (pantalla) o a un archivo o una impresora u otro nodo de internet.
- Los archivos son ejemplos de flujos de doble dirección, se puede recibir y se puede enviar información desde y hacia ellos.

En capítulos anteriores ya se había hecho uso de los flujos; con el objeto `cin` se recibía información desde el teclado y con el objeto `cout` se enviaba información a la consola de salida en la pantalla. En este capitulo, además de estudiar los flujos `cin` y `cout`, se tratará el tema de usar archivos como ejemplos de flujos de entrada y salida de información distintos al teclado y la consola.

9.2. La jerarquía del conjunto de los flujos

El conjunto de todos los flujos se representará mediante el símbolo

$$\mathcal{S}$$

el cual es una abreviación de la palabra *stream* que es flujo en inglés.

El conjunto $\mathcal{S}$ contiene a el conjunto $\mathcal{IOS}$ (*input output stream*) de los flujos que reciben o envían información, o que reciben y envían información, es decir,

$$\mathcal{IOS} \subseteq \mathcal{S}$$

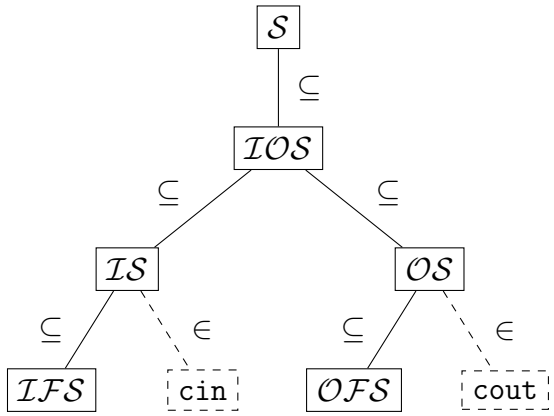
El conjunto $\mathcal{IOS}$ contiene a los conjuntos $\mathcal{IS}$ (*input stream*) y $\mathcal{OS}$ (*output stream*), es decir,

$$\mathcal{IS} \subseteq \mathcal{IOS} \qquad \text{y} \qquad \mathcal{OS} \subseteq \mathcal{IOS}$$

El conjunto $\mathcal{IS}$ a su vez contiene a el conjunto $\mathcal{IFS} \subseteq \mathcal{IS}$ (*input file stream*), y el objeto `cin` pertenece a él ($\texttt{cin} \in \mathcal{IS}$).

El conjunto $\mathcal{OS}$ a su vez contiene a el conjunto $\mathcal{OFS} \subseteq \mathcal{OS}$ (*output file stream*), y el objeto `cout` pertenece a él ($\texttt{cout} \in \mathcal{OS}$).

En el siguiente árbol se muestra la jerarquía de los flujos descrita anteriormente



9.3. Los flujos en C++

- Los flujos de entrada $\mathcal{IS}$ en C++ se codifican con la palabra `istream`.
- Los flujos de entrada desde un archivo $\mathcal{IFS}$ en C++ se codifican con la palabra `ifstream`.
- Los flujos de salida $\mathcal{OS}$ en C++ se codifican con la palabra `ostream`.
- Los flujos de salida a un archivo $\mathcal{OFS}$ en C++ se codifican con la palabra `ofstream`.
- Los flujos de entrada o salida de datos $\mathcal{IOS}$ en C++ se codifican con la palabra `iostream`.
- El objeto `cin` pertenece al conjunto $\mathcal{IS}$ que es un subconjunto de $\mathcal{IOS}$.
- El objeto `cout` pertenece al conjunto $\mathcal{OS}$ que es un subconjunto de $\mathcal{IOS}$.

La definición de flujos de entrada y salida simultanea de datos se encuentra en C++ en la librería `iostream`, por esto es necesario incluir al principio de las funciones que manejan flujos esta librería, así como se muestra en el siguiente encabezado de un archivo de un programa en C++

```
#include<iostream>
#include<cstdlib>

using namespace std;
```

Lectura de datos: si se quiere leer un dato de un tipo primitivo $\mathbb{T}$ desde un flujo de entrada $\mathcal{IS}$, entonces una función que representa este procedimiento se puede escribir de la siguiente forma

$$\begin{aligned} leer\_T : \mathcal{IS} &\to \mathbb{T} \\ (is) &\mapsto a, \quad \text{donde} \quad a = leer(is) \end{aligned}$$

La traducción de esta función en C++ se escribe de la siguiente manera

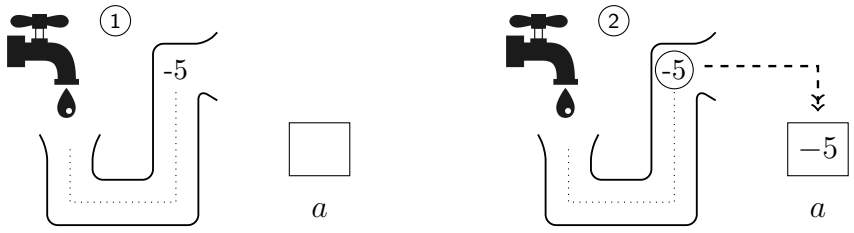
```
T* leer_T(istream& is){
   T a;
   is >> a;
   return a;
};
```

La función $leer(is)$ que en C++ se escribe como `>>`, ésta retorna el dato siguiente que se encuentre almacenado en el flujo is.

Cuando un flujo se utiliza como parámetro, este se pasa por referencia, es decir, que físicamente no se pasa un valor como parámetro, como en el caso de los tipos primitivos, sino que se pasa su dirección de memoria. Para pasar la dirección de memoria de un flujo se coloca el símbolo `&` inmediatamente después del tipo de flujo; como en el caso anterior que se escribio `istream&`.

Ejemplo. En C++ para leer un dato de tipo entero desde un flujo se utiliza la siguiente función

```
int leer_int(istream& is){
   int a;
   is >> a;
   return a;
};
```



Escritura de datos: si se quiere escribir un dato de un tipo de primitivo $\mathbb{T}$ en un flujo de salida $\mathcal{OS}$, entonces una función que representa este procedimiento se puede escribir de la siguiente forma

$$\begin{aligned} escribir\_T : \mathbb{T} \times \mathcal{OS} &\to \mathcal{OS} \\ (a, os) &\mapsto os, \quad \text{donde} \quad escribir(a, os) \end{aligned}$$

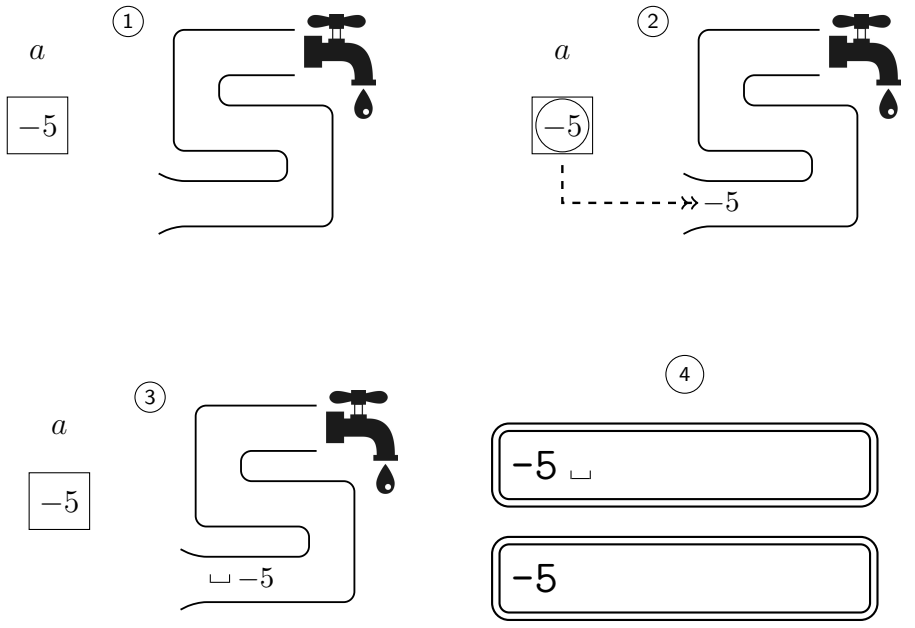
La traducción de esta función en C++ se escribe de la siguiente manera

```
ostream& escribir_T(T a, ostream& os){
    os << a;
    os << "\t";
    return os;
};
```

La función *escribir*(a, os) que en C++ se escribe como `<<`, ésta escribe el dato a en el flujo os. En la función `escribir_T` se envía adicionalmente el símbolo `"\t"` para separar el símbolo a del siguiente dato que sea almacenado en el flujo os.

Ejemplo. En C++ para escribir un dato de tipo entero en un flujo se utiliza la siguiente función

```
ostream& escribir_int(int a, ostream& os){
   os << a;
   os << "\t";
   return os;
};
```



9.3.1. Ejemplo del uso de los flujos de entrada y salida estándares

Ejemplo. Para utilizar las funciones anteriores, que leen desde el teclado y escriben en la consola se pueden llamar de la siguiente forma

```
int main(){
   cout << "Digite un entero: ";
   int a = leer_int(cin);
   cout << "El entero leido es: ";
   escribir_int(a, cout);
   cout << "\n";
   system("pause");
   return EXIT_SUCCESS;
};
```

Para el programa anterior se tiene como salida en la consola el siguiente texto

```
Digite un entero: -5
El entero leido es: -5
Presione una tecla para continuar . . .
```

9.4. Flujos de entrada y salida desde y hacia archivos

Para poder usar archivos como flujos de entrada o salida es necesario en primera instancia incluir la librería

```
#include<fstream>
```

con lo cual el encabezado de un archivo fuente de un programa en C++ que haga uso de archivos como flujos tendrá el siguiente aspecto

```
#include<iostream>
#include<cstdlib>
#include<fstream>

using namespace std;
```

9.4.1. Uso de archivos como flujos de entrada

Para declarar que un archivo f pertenece al flujo fuente de entrada $\mathcal{IFS}$ ($f \in \mathcal{IFS}$), en C++ se especifica que f pertenece al conjunto `ifstream` y se debe proporcionar la ubicación del archivo en el computador donde se ejecute el programa, es necesario que el archivo exista previamente.

Para crear un flujo de entrada en C++ y especificar la ruta de localización del archivo que sirve de flujo de entrada se utiliza la siguiente sintaxis

```
ifstream f("<path>");
```

el parámetro de la ruta `<path>` donde se ubica el archivo fuente de entrada depende del sistema operativo. Si se proporciona como localización `"archivo.txt"`, se abrirá el

archivo llamado `"archivo.txt"` que se encuentra en la misma carpeta (en la misma ruta) del archivo ejecutable.

Para proporcionar una ruta especifica de un archivo en Windows es necesario especificar la unidad y el directorio.

Ejemplo. Un ejemplo de la ruta de la ubicación de un archivo en Windows es

```
C:\\mis documentos\\archivo.txt
```

Para proporcionar una ruta especifica de un archivo en Linux es necesario especificarla desde el directorio raiz.

Ejemplo. Un ejemplo de la ruta de la ubicación de un archivo en Linux es

```
/home/user/archivo.txt
```

9.4.2. Uso de archivos como flujos de salida

Para declarar que un archivo f pertenece al flujo de salida $\mathcal{OFS}$ ($f \in \mathcal{OFS}$), en C++ se especifica que f pertenece al conjunto `ofstream` y se debe proporcionar la ubicación del archivo en el computador donde se ejecute el programa, en este caso no es necesario que el archivo exista previamente.

Para crear un flujo de salida en C++ y especificar la ruta de localización del archivo que sirve de flujo de salida se utiliza la siguiente sintaxis

```
ofstream f("<path>");
```

Las reglas para la especificación de la ruta de localización son las mismas que se describieron para los archivos como flujos de entrada.

9.4.3. Cierre de los flujos desde y hacia archivos

Una paso necesario cuando se utilizan archivos como flujos de entrada o salida es que es necesario cerrar los archivos después de utilizarlos para evitar que al ser abiertos queden bloqueados para ser usados por otras aplicaciones o que al escribir la información no se escriba completamente en el archivo. Para hacer esto se llama la función `close()` con la sintaxis

```
f.close();
```

donde `f` es el flujo que se creo con el archivo especificado.

9.4.4. Ejemplo del uso de archivos como flujos de entrada y salida

Ejemplo. Suponga que se dispone del archivo de texto plano `entrada.txt` con las siguiente información

```
0
1
2
3
4
5
```

y se quiere leer de este archivo para copiar los primeros 4 enteros del archivo al archivo `salida.txt`, esto se podría hacer utilizando el siguiente programa

```
#include<iostream>
#include<cstdlib>
#include<fstream>

using namespace std;
```

```
int leer_int(istream& is){
   int a;
   is >> a;
   return a;
};
```

```
ostream& escribir_int(int a, ostream& os){
   os << a;
   os << "\t";
   return os;
};
```

```
int main(){
  ifstream ifs("entrada.txt");
  ofstream ofs("salida.txt");
  for(int i = 0; i < 4; i++){
      escribir_int(leer_int(ifs), ofs);
  };
  ofs.close();
  ifs.close();
  cout << "\n";
  system("pause");
  return EXIT_SUCCESS;
};
```

La salida de dicho programa será:

```
Presione una tecla para continuar . . .
```

y se generará el archivo `salida.txt` con el siguiente contenido:

```
0    1    2    3
```

Esta declaración de flujo sobreescribe el archivo cada vez que se ejecuta el programa. Nótese que cada dato con esta sintaxis se lee hasta encontrar uno de los siguientes caracteres especiales: `'\t'`, `'\n'`, `' '`. La primera vez que se entra en el ciclo `for` se lee el entero `0`, luego se realiza la lectura del `1`, luego se realiza la lectura del `2` y finalmente se lee el entero `3`, cada uno de esos números se escriben en el archivo de salida junto con un símbolo `'\t'` y a continuación se procede a cerrar los flujos.

Ejercicios

Capítulo 10

Vectores o arreglos unidimesionales

10.1. Conceptos y notación

Un *vector* es una n-tupla de n objetos llamados las *componentes* del vector, los cuales generalmente son números, caracteres, valores booleanos, y cualesquiera otro tipo de elementos que pertenecen a un mismo conjunto.,

Ejemplo. La siguiente quíntupla de números enteros denotada por la expresión v es un vector

$$v = (1, 0, 7, -2, 8)$$

En general, un vector v se puede representar de la siguiente forma

$$v = (v_1, v_2, v_3, \ldots, v_n)$$

donde el vector está constituido por n componentes de un conjunto genérico $\mathcal{V}$. Si $v \in \mathcal{V}^n$, entonces el vector se dice que es n-dimensional o de tamaño n.

Ejemplo. El siguiente vector de tamaño 6 pertenece a $\mathbb{Z}^6$ y tiene una notación particular la cual es $\mathbf{0}_6$

$$\mathbf{0}_6 = (0, 0, 0, 0, 0, 0)$$

En un vector, para referirse a una componente en particular, a ésta se le dice que es la *componente* en la posición i o la i-ésima componente, esto significa que el objeto es la componente ubicada en la posición i, se denota por la expresión v_i y se puede ubicar dentro del vector como se muestra a continuación

$$\begin{array}{c} \text{componente } i\text{-ésima} \\ \downarrow \\ (v_1, \ldots, v_i, \ldots, v_n) \end{array}$$

Ejemplo. Para el vector

$$v = \Big(-1,\ \tfrac{3}{4},\ -0.25,\ -\tfrac{1}{5},\ \sqrt{2},\ 0.0,\ \pi,\ \sqrt[3]{5},\ 0.\overline{9} \Big)$$

de $\mathbb{R}^9$ se tiene que sus componentes son:

- $v_1 = -1$.
- $v_2 = \frac{3}{4}$.
- $v_3 = -0.25$.
- $v_4 = -\frac{1}{5}$.
- $v_5 = \sqrt{2}$.
- $v_6 = 0.0$.
- $v_7 = \pi$.
- $v_8 = \sqrt[3]{5}$.
- $v_9 = 0.\overline{9}$.

10.1.1. El conjunto de los vectores

A partir de la notación de producto generalizado de un conjunto $\mathcal{V}$

$$\mathcal{V}^n = \underbrace{\mathcal{V} \times \mathcal{V} \times \mathcal{V} \times \cdots \times \mathcal{V}}_{n\text{-veces}}$$

se puede obtener el conjunto de los vectores $\mathcal{V}^*$, el cual se define como la unión de todos los productos cartesianos del conjunto $\mathcal{V}$, de la siguiente manera

$$\mathcal{V}^* = \bigcup_{n \in \mathbb{N}} \mathcal{V}^n$$

10.2. Los arreglos o vectores en computación

Si se tiene un conjunto $\mathbb{T}$ que representa un tipo de datos y un número $n \in \mathbb{N}$ se puede construir un vector de tamaño n, a los vectores que se construyen sobre tipos de datos en computación se les llama *arreglos* o *vectores unidimensionales*.

A partir del producto generalizado de un conjunto $\mathbb{T}$ se puede obtener el conjunto de los arreglos $\mathbb{T}^*$, el cual se define como la unión de todos los productos cartesianos del conjunto $\mathbb{T}$, de la siguiente manera

$$\mathbb{T}^* = \bigcup_{n \in \mathbb{N}} \mathbb{T}^n \quad \text{y se llama el conjunto de todos los arreglos de tipo } \mathbb{T}.$$

así, el conjunto de los arreglos del tipo de datos $\mathbb{T}$ es una colección de variables del tipo de datos $\mathbb{T}$ que están subindicadas, esto es, que se accede a ellas por medio de un índice que específica una componente particular del arreglo.

Dado un arreglo $\mathtt{x} \in \mathbb{T}^*$, para acceder en C++ a la variable almacenada en la componente i se utiliza la notación

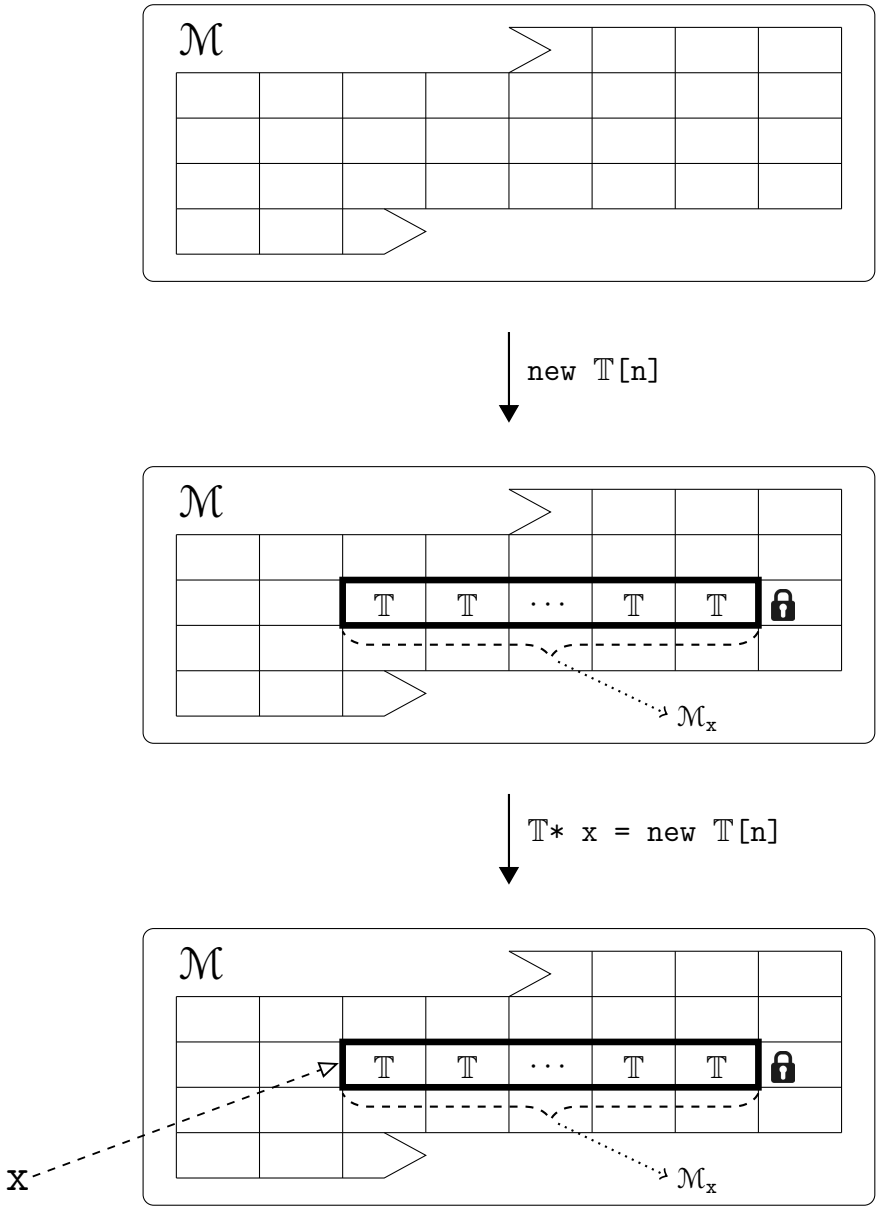
$$\mathtt{x}_i \equiv \mathtt{x[i]}$$

Para definir arreglos se utilizará la notación de memoria dinámica, es decir, si $\mathtt{x} \in \mathbb{T}^*$ entonces el vector `x` se creará en tiempo de ejecución del programa.

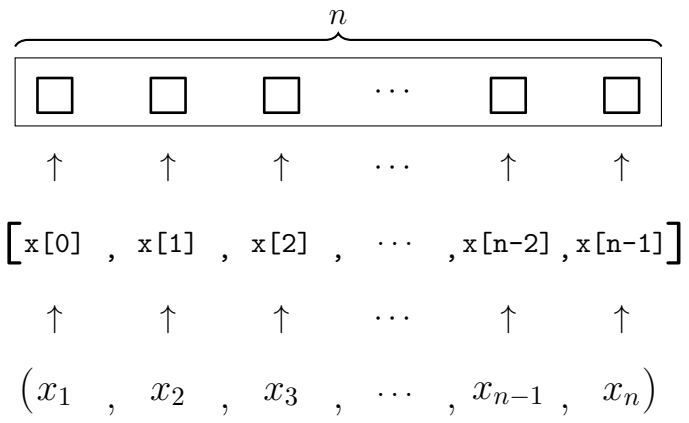
Si se quiere expresar en C++ que $\texttt{x} \in \mathbb{T}^*$ esto se escribe como `T* x`; y para reservar el espacio de memoria para todo el arreglo de tipo $\mathbb{T}$, esto se escribe como `x = new T[n]`. De donde, para crear un arreglo `x` de tamaño n y de tipo $\mathbb{T}$ se utiliza la instrucción

```
T* x = new T[n]
```

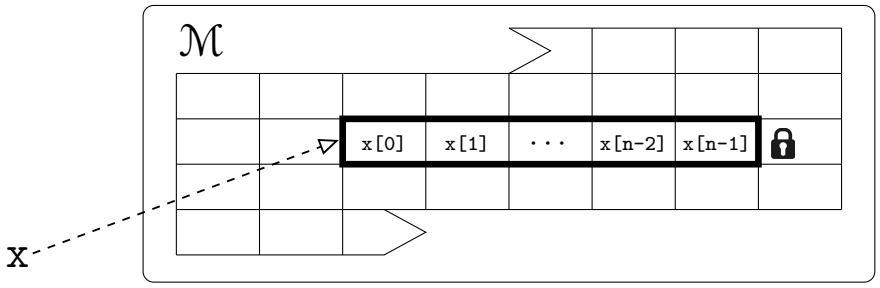
Con esta instrucción se reserva una porción de la memoria $\mathcal{M}$ que es utilizada para almacenar el arreglo y que es bloqueada para que sólo se pueda utilizar para guardar valores en el arreglo, a menos que el espacio se libere y se pueda utilizar para almacenar otro tipo de información. A la porción de espacio de memoria que ocupa un arreglo `x` lo notaremos como $\mathcal{M}_{\texttt{x}}$. Gráficamente esto se puede ver así:



Cuando se define un arreglo en C++, es necesario tener en cuenta que la primera posición del arreglo es la ubicada en la componente 0, y para un arreglo de tamaño n se tiene que la posición máxima del arreglo será la ubicada en la componente $n-1$. Es decir, para un vector $x = (x_1, x_2, x_3, \ldots, x_{n-1}, x_n)$ la representación del arreglo en C++ es la que se presenta en la siguiente figura.



con lo cual, la representación en la memoria resultará ser



En caso de que se quiera acceder a una componente mayor o igual a la n-ésima, entonces el valor almacenado será lo que en ese momento tenga la memoria en el siguiente campo de la memoria, pero éste será un valor inesperado, por que simplemente no se ha protegido ese espacio de memoria y la información allí es desconocida a priori. Si se quiere acceder a una componente menor a 0, ocurre algo similar a lo descrito anteriormente.

10.2.1. Funciones para utilizar arreglos

Para crear y eliminar arreglos se diseñarán las respectivas funciones de creación y de eliminación.

10.2.1.1. Creación de arreglos

Matemáticamente se definirá una rutina de creación de un arreglo como aquella rutina que dado un $n \in \mathbb{N}$, correspondiente al tamaño, retornará un elemento de $\mathbb{T}^*$ que es de tamaño n.

$$crear\_arreglo\_T : \mathbb{N} \to \mathbb{T}^*$$
$$(n) \mapsto x, \quad \text{donde } x \in \mathbb{T}^n \subsetneq \mathbb{T}^*$$

En C++ se traduce

```
T* crear_arreglo_T(int n){
   return new T[n];
};
```

Ejemplo. Para crear un arreglo de tipo entero se tiene la siguiente función

```
int* crear_arreglo_int(int n){
   return new int[n];
};
```

10.2.1.2. Eliminación de arreglos

Para eliminar un arreglo, se debe entender que lo que ocurre es que la porción del espacio de la memoria que se utiliza para almacenar las componentes del arreglo se liberan o se regresan al sistema operativo para que éste disponga de esa memoria para almacenar nueva información.

Matemáticamente se puede modelar esto como una función que dado el arreglo x y el tamaño del arreglo n (aunque este último no se requiere, pero se suele utilizar), se retorna la porción del espacio de la memoria $\mathcal{M}_x$ que es utilizado para almacenar el arreglo x, de esta manera la función se puede escribir así

$$liberar\_arreglo\_T : \mathbb{T}^* \times \mathbb{N} \to \wp(\mathcal{M})$$
$$(x, n) \mapsto \mathcal{M}_x$$

Hay que recordar aquí que $\wp(\mathcal{M})$ es el conjunto de todos los subconjuntos de la memoria $\mathcal{M}$ y que dado un arreglo x, $\mathcal{M}_x$ es una porción de memoria donde se almacena el arreglo x, por lo tanto $\mathcal{M}_x \subseteq \mathcal{M}$, es decir $\mathcal{M}_x \in \wp(\mathcal{M})$.

Para traducir esta función a C++ se debe tener en cuenta que la función no retorna un valor de un tipo de datos, si no que se regresa memoria al sistema operativo, por lo que la función regresa un espacio que esta vació y queda disponible para utilizarse de nuevo.

El desbloqueo de la memoria y la liberación del espacio utilizado por el arreglo x se escribe en C++ mediante la instrucción

```
delete[] x;
```

Para decir que se retorna memoria vacía al sistema operativo se utiliza como tipo de dato el vacío que en C++ se escribe como `void` y en el cuerpo de la función se utiliza la instrucción `return;` para indicar que se retorno una porción de espacio que esta vació y que queda disponible para usarse de nuevo.

Una función que sólo manipula la memoria y que retorna un tipo de dato vacío, en programación se conoce como un *procedimiento*.

De esta manera, la traducción de la función en C++ se escribe de la siguiente manera

```
void liberar_arreglo_T(T* x, int n){
   delete[] x;
   return;
};
```

Ejemplo. Para liberar la memoria usada por un arreglo de tipo carácter se tiene la siguiente función

```
void liberar_arreglo_char(char* x, int n){
   delete[] x;
   return;
};
```

10.3. Arreglos y flujos de datos

Dado un arreglo de tipo $\mathbb{T}$, es posible realizar operaciones de lectura y escritura sobre flujos de datos, y dichas operaciones se definen así

Lectura de matrices: para la entrada o lectura de un arreglo desde un flujo de datos se define la siguiente función

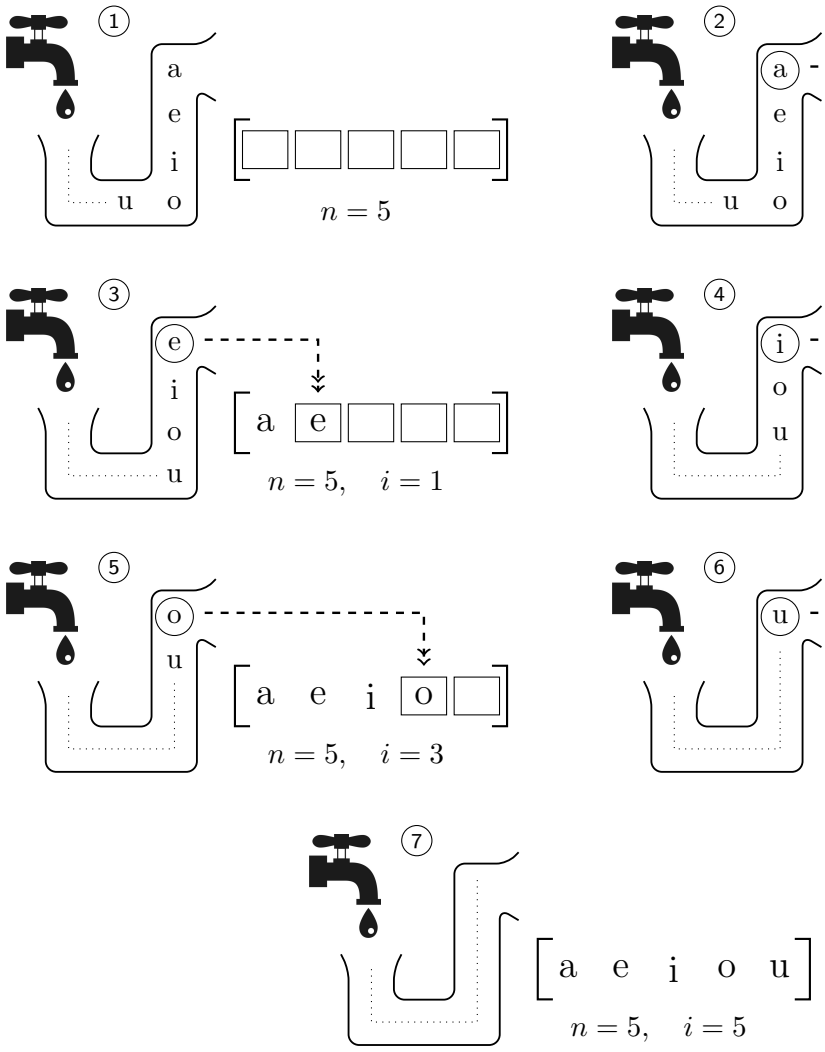
$$\begin{aligned} leer\_arreglo\_T : \mathcal{IS} \times \mathbb{T}^* \times \mathbb{N} &\to \mathbb{T}^* \\ (is, x, n) &\mapsto x, \qquad \text{donde} \quad x_i = leer\_T(is), \\ & \qquad\qquad\qquad\quad \forall i = 1, 2, 3, \ldots, n. \end{aligned}$$

La traducción de esta función en C++ se escribe de la siguiente manera (es necesario tener en cuenta que como se mencionó previamente, los arreglos en C++ comienzan en la posición 0):

```
T* leer_arreglo_T(istream& is, T* x, int n){
  for(int i = 0; i < n; i++){
    is >> x[i];
  };
  return x;
};
```

Ejemplo. En C++ para leer un arreglo de tipo carácter se utiliza la siguiente función

```
char* leer_arreglo_char(istream& is, char* x, int n){
  for(int i = 0; i < n; i++){
    is >> x[i];
  };
  return x;
};
```



Escritura de arreglos: para enviar o escribir un arreglo en un flujo de datos se define la siguiente función

$$\begin{aligned} \mathit{escribir\_arreglo\_T} : \mathcal{OS} \times \mathbb{T}^* \times \mathbb{N} &\to \mathcal{OS} \\ (os, x, n) &\mapsto os, \qquad \text{donde} \quad \mathit{escribir\_T}(x_i, os), \\ & \qquad\qquad\qquad\quad \forall i = 1, 2, 3, \ldots, n. \end{aligned}$$

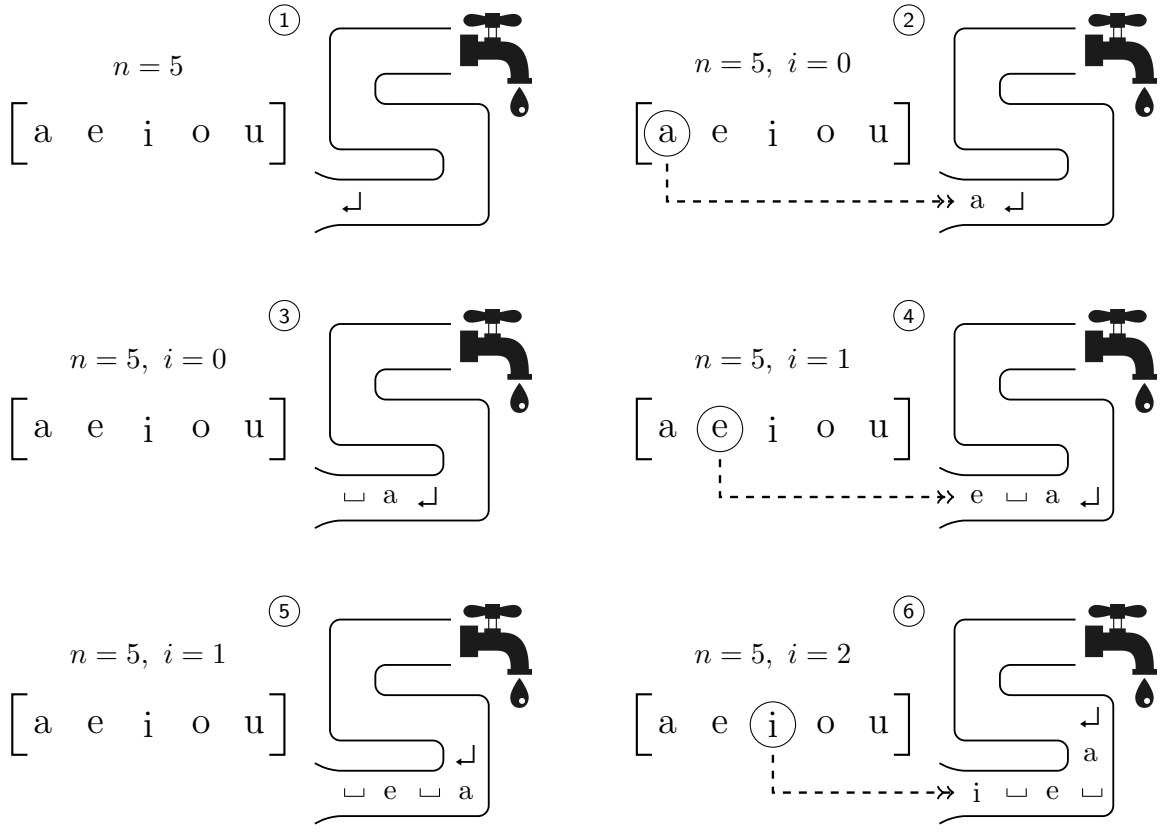
La traducción de esta función en C++ se escribe de la siguiente manera. En primera instancia se inicia una nueva línea y para que los datos se puedan diferenciar al leerlos de

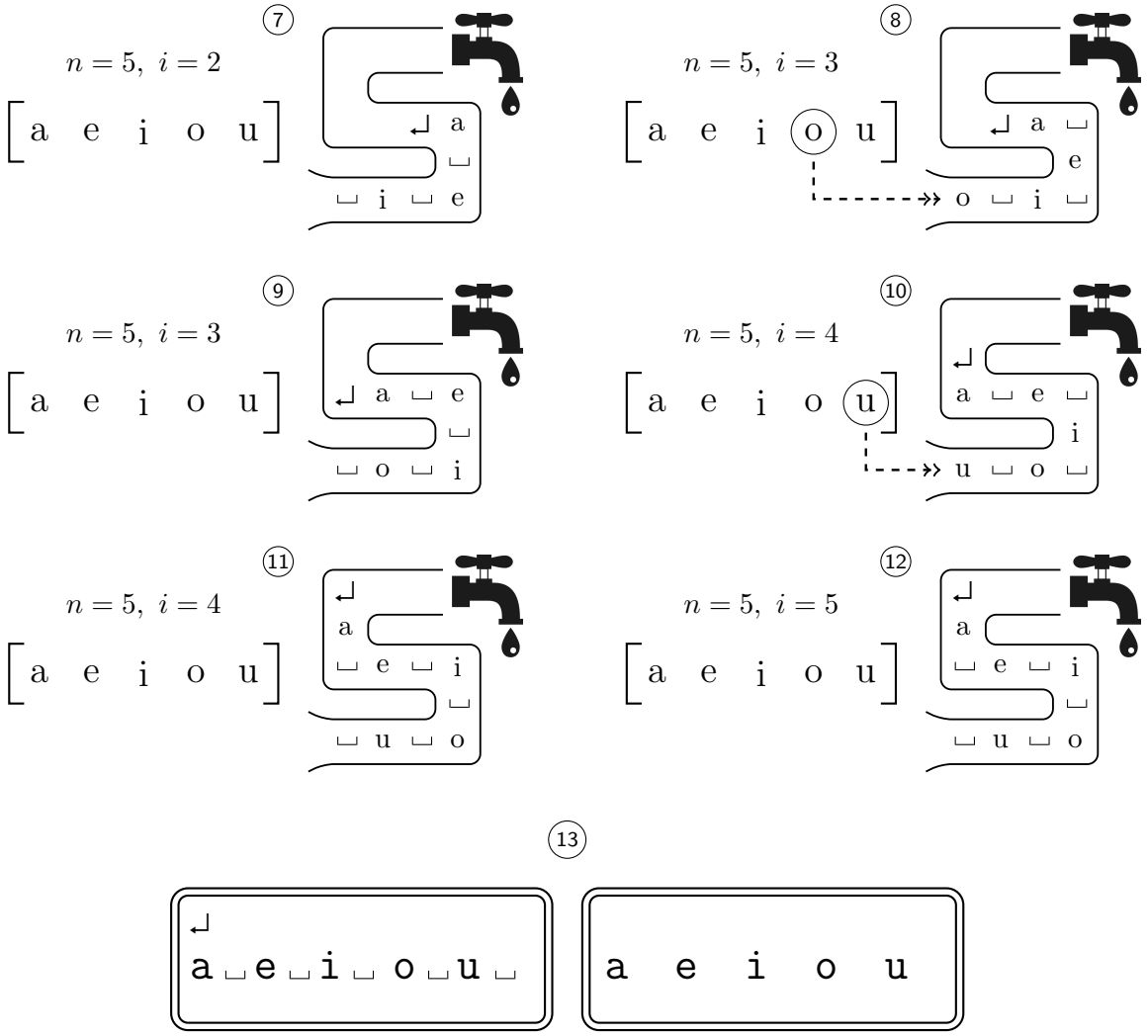
nuevo, se escribe un símbolo de tabulación o un espacio en blanco cada vez que se escribe un nuevo dato en el flujo de datos

```
ostream& escribir_arreglo_T(ostream& os, T* x, int n){
  os << "\n";
  for(int i = 0; i < n; i++){
    os << x[i];
    os << "\t";
  };
  return os;
};
```

Ejemplo. En C++ para escribir un arreglo de tipo carácter se utiliza la siguiente función

```
ostream& escribir_arreglo_char(ostream& os, char* x, int n){
  os << "\n";
  for(int i = 0; i < n; i++){
    os << x[i];
    os << "\t";
  };
  return os;
};
```





10.3.1. Ejemplos de funciones con arreglos

Es posible utilizar lo visto en funciones para realizar diversidad de cálculos que involucren arreglos.

Ejemplo. *El cubo de las componentes de arreglos numéricos enteros*

Suponga que un archivo contiene unos datos numéricos enteros tales que consta de 5 datos que se encuentran separados por una tabulación así como se muestra a continuación

1	2	3	4	5

Una función general que permite construir un nuevo arreglo que contiene el cubo de cada componente de un arreglo dado es

$$\begin{aligned} &cubo\_arreglo : \mathbb{Z}^* \times \mathbb{N} \to \mathbb{Z}^* \\ &\qquad (x, n) \mapsto y, \qquad \text{donde} \quad y_i = x_i^3, \\ &\qquad\qquad\qquad\qquad\quad \forall i = 1, 2, \ldots, n. \end{aligned}$$

Un programa completo en C++ que permite calcular el cubo de las componentes de un arreglo obtenido a partir del archivo presentado anteriormente es

```
#include<iostream>
#include<cstdlib>
#include<fstream>

using namespace std;
```

```
int* crear_arreglo_int(int n){
   return new int[n];
};
```

```
void liberar_arreglo_int(int* x, int n){
   delete[] x;
   return;
};
```

```
int* leer_arreglo_int(istream& is, int* x, int n){
   for(int i = 0; i < n; i++){
      is >> x[i];
   };
   return x;
};
```

```
ostream& escribir_arreglo_int(ostream& os, int* x, int n){
   os << "\n";
   for(int i = 0; i < n; i++){
      os << x[i];
      os << "\t";
   };
   return os;
};
```

```
int* cubo_arreglo(int* x, int n){
   int* y = crear_arreglo_int(n);
   for(int i = n - 1; i >= 0; i--){
      y[i] = x[i] * x[i] * x[i];
   };
   return y;
};
```

```
int main(){
   int n = 5;
   ifstream ifs("arreglo_numeros.txt");
   ofstream ofs("arreglo_cubos.txt");
   int* x = crear_arreglo_int(n);
   x = leer_arreglo_int(ifs, x, n);
   int* y = cubo_arreglo(x, n);
   escribir_arreglo_int(ofs, y, n);
   liberar_arreglo_int(x, n);
   liberar_arreglo_int(y, n);
   ifs.close();
   ofs.close();
   cout  << "El calculo del arreglo fue exitoso\n";
   system("pause");
   return  EXIT_SUCCESS;
};
```

El archivo donde se almacena el resultado de ejecutar el anterior programa tiene el siguiente aspecto

```
1     8     27     64     125
```

Una función recursiva que permite también calcular el cubo de las componentes de un arreglo mediante la creación de un nuevo arreglo que almacene el resultado del cálculo, y una función ayudante o auxiliar donde se realiza el cálculo componente a componente y se hacen los llamados recursivos para recorrer el arreglo original. Estas funciones pueden ser escritas de la siguiente manera:

```
int* cubo_arreglo_aux(int* x, int* y, int n){
   y[n - 1] = x[n - 1] * x[n - 1] * x[n - 1];
   if(n == 1){
      return y;
   }else{
      return cubo_arreglo_aux(x, y, n - 1);
   };
};
```

```
int* cubo_arreglo(int* x, int n){
   int* y = crear_arreglo_int(n);
   return cubo_arreglo_aux(x, y, n);
};
```

Ejemplo. *La suma de los elementos de un arreglo de números reales*

Supónga que se solicita calcular la suma de las componentes numéricas de un arreglo de tipo real. Un posible modelo matemático puede ser obtenido teniendo en cuenta que:

- Si el arreglo tiene una única componente, entonces el valor de la suma es el valor de la componente.
- Si el arreglo tiene más de una componente, entonces se pueden sumar recursivamente las primeras componentes del subarreglo que no contiene la última componente, y a continuación se adiciona esta última componente del arreglo. Así, la suma total se obtiene sumando las componentes de arreglos cuya longitud va decreciendo durante el cálculo hasta llegar a una longitud igual a 1.

Una posible definición teniendo en cuenta las observaciones anteriores y que el segundo parámetro de la función representa la longitud del arreglo se presenta a continuación

$$suma\_arreglo : \mathbb{R}^* \times \mathbb{N} \to \mathbb{R}$$
$$(x, n) \mapsto \begin{cases} x_1, & \text{si} \quad n = 1; \\ suma\_arreglo(x, n-1) + x_n, & \text{en otro caso.} \end{cases}$$

La codificación en C++ de una función que permite sumar las componentes de un arreglo de números reales de longitud n se presenta a continuación. Al codificar la función en C++ es necesario tener en cuenta que las componentes sufren un corrimiento de una posición, pero la llamada recursiva de la función se hace igual que como fue definida en el modelo matemático.

```
double suma_arreglo(double* x, int n){
   if(n == 1){
      return x[0];
   };
   return suma_arreglo(x, n - 1) + x[n-1];
};
```

Una función en C++ que calcula la suma de las componentes de un arreglo, que está escrita usando estructuras cíclicas y que resulta ser equivalente a la anterior es la siguiente

```
double suma_arreglo(double* x, int n){
   double S = x[0];
   for(int i = 1; i < n; i++){
      S = S + x[i];
   };
   return S;
};
```

Ejemplo. *El promedio de las componentes de un arreglo de reales*

Se puede tomar el resultado de la función anterior para calcular el promedio de los elementos de un arreglo utilizando el siguiente modelo de función matemática

$$promedio : \mathbb{R}^* \times \mathbb{N} \to \mathbb{R}$$
$$(x, n) \mapsto suma\_arreglo(x, n)/n;$$

Una función en C++ que permite calcular el promedio de los datos numéricos almacenados en un arreglo de longitud n es

```
double promedio(double* x, int n){
   return suma_arreglo(x, n) / n;
};
```

Ejemplo. *La conjetura de los arreglos de números reales ordenados ascendentemente*

Un arreglo está ordenado ascendentemente si $x_i \leq x_{i+1}, \forall i = 1, 2, \ldots, n-1$. Una función recursiva que permite determinar si un arreglo se encuentra ordenado ascendentemente se puede modelar matemáticamente teniendo en cuenta que:

- Si el arreglo tiene una única componente, entonces el arreglo se encuentra ordenado ascendentemente.
- Si el arreglo tiene más de una componente, entonces, si el subarreglo que no contiene la última componente del arreglo original está ordenado ascendentemente, basta con establecer que la última componente del arreglo original es mayor o igual a la última componente del subarreglo.

$$esta\_ordenado : \mathbb{R}^* \times \mathbb{N} \to \mathbb{B}$$
$$(x, n) \mapsto \begin{cases} V, & \text{si} \quad n = 1; \\ F, & \text{si} \quad x_{n-1} > x_n; \\ esta\_ordenado(x, n-1), & \text{en otro caso.} \end{cases}$$

La codificación en C++ de una función que permite determinar si un arreglo de longitud n se encuentra ordenado ascendentemente se presenta a continuación. Al codificar la función en C++ es necesario tener en cuenta que las componentes sufren un corrimiento de una posición, pero la llamada recursiva de la función se hace igual que como fue definida en el modelo matemático.

```
bool esta_ordenado(double* x, int n){
   if(n == 1){
      return true;
   };
   if(x[n - 2] > x[n - 1]){
      return false;
   };
   return esta_ordenado(x, n - 1);
};
```

Una función en C++ que permite determinar si un arreglo se encuentra ordenado ascendentemente, que está escrita usando estructuras cíclicas y que resulta ser equivalente a la anterior es la siguiente

```
bool esta_ordenado(double* x, int n){
   for(int i = n - 2; i >= 0; i--){
      if(x[i] > x[i + 1]){
         return false;
      };
   };
   return true;
};
```

Ejemplo. *El máximo de un subarreglo de tipo real*

El máximo de un arreglo es un valor M que pertenece al arreglo tal que $M \geq x_i$, $\forall i = 1, 2, \ldots, n$. Una función recursiva que permite calcular el máximo de un arreglo se puede modelar matemáticamente teniendo en cuenta que:

- Si el arreglo tiene una única componente, entonces el máximo del arreglo es igual al valor de esa componente.

- Si el arreglo tiene más de una componente, entonces, si el máximo del subarreglo que no contiene la última componente del arreglo original se compara con última componente del arreglo original, entonces el mayor de los dos resulta ser el mayor de arreglo original.

$$max\_arreglo : \mathbb{R}^* \times \mathbb{N} \to \mathbb{R}$$

$$(x, n) \mapsto \begin{cases} x_1, & \text{si} \quad n = 1; \\ M, & \text{donde} \quad \big(M = max\_arreglo(x, n-1)\big) \wedge \\ & \qquad\quad (M > x_n); \\ x_n, & \text{en otro caso.} \end{cases}$$

La codificación en C++ de una función que permite calcular el máximo de un arreglo de longitud n se presenta a continuación. Al codificar la función en C++ es necesario tener en cuenta que las componentes x_n sufren un corrimiento de una posición desde el intervalo original n a 1 al intervalo trasladado desde $n-1$ hasta 0, pero la llamada recursiva de la función se hace igual que como fue definida en el modelo matemático.

```
double max_arreglo(double* x, int n){
   if(n == 1){
      return x[0];
   };
   double M = max_arreglo(x, n - 1);
   if(M > x[n - 1]){
      return M;
   };
   return x[n - 1];
};
```

Una función en C++ que permite calcular el máximo de un arreglo, que está escrita usando estructuras cíclicas y que resulta ser equivalente a la anterior es la siguiente

```
double max_arreglo(double* x, int n){
   double M = x[0];
   for(int i = 1; i < n; i++){
      if(M < x[i]){
         M = x[i];
      };
   };
   return M;
};
```

Ejemplo. *La posición del máximo de un subarreglo de números reales*

La posición del máximo de un arreglo es un valor k tal que $1 \leq k \leq n$ y $x_k \geq x_i$, $\forall i = 1, 2, \ldots, n$. Una función recursiva que permite calcular la posición del máximo de un arreglo se puede modelar matemáticamente teniendo en cuenta que:

- Si el arreglo tiene una única componente, entonces la componente del máximo se encuentra ubicada en ésta primera posición.
- Si el arreglo tiene más de una componente, entonces, la posición del máximo se obtiene calculando la posición del máximo del subarreglo que no contiene la última componente del arreglo original y si al comparar esa componente con última componente del arreglo original, entonces la posición del mayor será la posición del máximo del arreglo original.

La codificación en C++ de una función que permite obtener la posición del máximo de un arreglo de longitud n se presenta a continuación

$$pos\_max : \mathbb{R}^* \times \mathbb{N} \to \mathbb{N}$$

$$(x, n) \mapsto \begin{cases} 1, & \text{si} \quad n = 1; \\ k, & \text{donde} \quad \big(k = pos\_max(x, n-1)\big) \wedge \\ & \qquad \big(x_k > x_n\big); \\ n, & \text{en otro caso.} \end{cases}$$

La codificación en C++ de una función que permite obtener la posición del máximo de un arreglo de longitud n se presenta a continuación.

```
int pos_max(double* x, int n){
   if(n == 1){
      return 0;
   };
   int k = pos_max(x, n - 1);
   if(x[k] > x[n - 1]){
      return k;
   };
   return n - 1;
};
```

Una función en C++ permite obtener la posición del máximo de un arreglo, que está escrita usando estructuras cíclicas y que resulta ser equivalente a la anterior es la siguiente

```
int pos_max(double* x, int n){
   int k = 0;
   for(int i = 1; i < n; i++){
      if(x[k] < x[i]){
         k = i;
      };
   };
   return k;
};
```

Ejemplo. *Ordenar ascendentemente un arreglo de reales por selección del máximo*

A partir de un arreglo $x = [x_1, x_2, \ldots, x_{n-1}, x_n]$ se puede obtener un nuevo arreglo $x' = [x'_1, x'_2, \ldots, x'_{n-1}, x'_n]$ tal que $\{x_1, x_2, \ldots, x_{n-1}, x_n\} = \{x'_1, x'_2, \ldots, x'_{n-1}, x'_n\}$ y el arreglo x' se encuentra ordenado ascendentemente, es decir $x'_1 \leq x'_2 \leq \cdots \leq x'_{n-1} \leq x'_n$. Una función recursiva que permite calcular la posición del máximo de un arreglo se puede modelar matemáticamente teniendo en cuenta que:

- Si el arreglo tiene una única componente, entonces el arreglo ya se encuentra ordenado ascendentemente.
- Si el arreglo tiene más de una componente, entonces, para ordenar un arreglo se puede utilizar el método de selección, en este caso se intercambia el valor máximo del arreglo con la componente que se encuentra en el último elemento del arreglo, y esto se realiza con cada subarreglo que no contiene la última componente del arreglo original.

La codificación en C++ de una función que permite obtener un arreglo ordenado ascendentemente a partir de un arreglo dado de longitud n se presenta a continuación.

Esta función hace un llamado a la función que permite obtener la posición del máximo de un arreglo dado y éste se utiliza para intercambiar el elemento mayor a la última posición, para realizar esto es necesario utilizar una variable adicional o auxiliar t que se utiliza para almacenar momentáneamente el valor de una de las componentes mientras ésta cambia su valor por el de la otra componente, y el valor de la otra componente cambia al valor almacenado en la variable t. Este procedimiento muy común al ordenar arreglos se conoce en programación como un *swap* de variables.

$$ordenar : \mathbb{R}^* \times \mathbb{N} \to \mathbb{R}^*$$

$$(x,n) \mapsto \begin{cases} x, & \text{si} \quad n = 1; \\ ordenar(x, n-1), & \text{donde} \quad \big(k = pos\_max(x,n)\big) \wedge \\ & \qquad (t = x_k) \ \wedge (x_k = x_n) \wedge (x_n = t). \end{cases}$$

La codificación en C++ de una función que permite obtener un arreglo ordenado ascendentemente a partir de un arreglo dado de longitud n se presenta a continuación.

```
double* ordenar(double* x, int n){
  if(n == 1){
     return x;
  }else{
     int k = pos_max(x, n);
     double t = x[k]; //swap 1
     x[k] = x[n-1];   //swap 2
     x[n-1] = t;      //swap 3
     return ordenar(x, n-1);
  };
};
```

Una función en C++ que permite obtener un arreglo ordenado ascendentemente a partir de un arreglo dado, que está escrita usando estructuras cíclicas y que resulta ser equivalente a la anterior es la siguiente

```
double* ordenar(double* x, int n){
   for(int i = n; i >= 1; i--){
      int k = pos_max(x, i);
      double t = x[k];
      x[k] = x[i - 1];
      x[i - 1] = t;
   };
   return x;
};
```

Ejercicios

Capítulo 11

Matrices o arreglos bidimensionales

11.1. Conceptos y notación

Una *matriz* es un arreglo rectangular de objetos, los cuales generalmente son números, caracteres, valores booleanos, y cualesquiera otro tipo de elementos que pertenecen a un mismo conjunto.

Ejemplo. La siguiente estructura rectangular de números enteros denotada por la expresión X es una matriz

$$X = \begin{bmatrix} 1 & 3 & 7 & -2 & 8 \\ 9 & 11 & 5 & 6 & 4 \\ 6 & -2 & -1 & 1 & 1 \end{bmatrix}$$

En general, una matriz X se puede representar de la siguiente manera

$$X = \begin{bmatrix} x_{11} & x_{12} & x_{13} & \cdots & x_{1(m-1)} & x_{1m} \\ x_{21} & x_{22} & x_{23} & \cdots & x_{2(m-1)} & x_{2m} \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ x_{(n-1)1} & x_{(n-1)2} & x_{(n-1)3} & \cdots & x_{(n-1)(m-1)} & x_{(n-1)m} \\ x_{n1} & x_{n2} & x_{n3} & \cdots & x_{n(m-1)} & x_{nm} \end{bmatrix}$$

donde la matriz está compuesta por n *filas* y m *columnas*, a esta matriz se le dice que es de *tamaño* $n \times m$.

$$X = \underbrace{\begin{bmatrix} x_{11} & x_{12} & \cdots & x_{1m} \\ x_{21} & x_{22} & \cdots & x_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ x_{n1} & x_{n2} & \cdots & x_{nm} \end{bmatrix}}_{m} \Big\} n$$

Ejemplo. La siguiente matriz es una matriz de tamaño 4×5

$$X = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

A los objetos de la matriz se les llaman *componentes* o *entradas* de la matriz, y para referirse a una componente en particular, a ésta se le dice que es la *componente* en la posición (i, j), esto significa que el objeto es la componente ubicada en la fila i y en el columna j, se denota por la expresión X_{ij} y se puede ubicar dentro de la matriz como se muestra a continuación

$$\begin{array}{c} \\ \\ \text{fila } i \longrightarrow \\ \\ \\ \end{array} \begin{array}{c} \text{columna } j \\ \downarrow \\ \begin{bmatrix} x_{11} & \cdots & x_{1j} & \cdots & x_{1m} \\ \vdots & \ddots & \vdots & \ddots & \vdots \\ x_{i1} & \cdots & \boxed{x_{ij}} & \cdots & x_{im} \\ \vdots & \ddots & \vdots & \ddots & \vdots \\ x_{n1} & \cdots & x_{nj} & \cdots & x_{nm} \end{bmatrix} \end{array}$$

Ejemplo. Para la matriz

$$X = \begin{bmatrix} 2 & -1 & \frac{3}{4} & -0.25 & e \\ -\frac{1}{5} & \sqrt{2} & 4 & 0.0 & 6 \\ 3.14 & \pi & \frac{1}{2} & \sqrt{3} & 3 \\ \sqrt[3]{5} & -10 & 0 & -5 & 0.\overline{9} \end{bmatrix}$$

de tamaño 4×5 se tiene que sus componentes son:

- $X_{11} = 2$.
- $X_{12} = -1$.
- $X_{13} = \frac{3}{4}$.
- $X_{14} = -0.25$.
- $X_{15} = e$.
- $X_{21} = -\frac{1}{5}$.
- $X_{22} = \sqrt{2}$.
- $X_{23} = 4$.
- $X_{24} = 0.0$.
- $X_{25} = 6$.
- $X_{31} = 3.14$.
- $X_{32} = \pi$.
- $X_{33} = \frac{1}{2}$.
- $X_{34} = \sqrt{3}$.
- $X_{35} = 3$.
- $X_{41} = \sqrt[3]{5}$.
- $X_{42} = -10$.
- $X_{43} = 0$.
- $X_{44} = -5$.
- $X_{45} = 0.\overline{9}$.

Cuando en una matriz se tiene que el número de filas es igual al número de columnas se dice que la matriz es cuadrada.

Ejemplo. La siguiente matriz de tamaño 2×2, es una matriz cuadrada cuyas entradas son valores del conjunto booleano ($\mathbb{B}$)

$$X = \begin{bmatrix} V & F \\ F & V \end{bmatrix}$$

Ejemplo. La siguiente matriz de tamaño 4×4, es una matriz cuadrada cuyas entradas son números reales, se le conoce como la *matriz identidad* de tamaño 4×4 y se denota por la expresión $\mathbf{I}_4$

$$\mathbf{I}_4 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

11.2. Definiciones alternativas

Una forma de entender la estructura interna de una matriz distinta a la definida previamente, es la de interpretarla como un arreglo de arreglos, esto es, verla como un arreglo cuyas componentes son a su vez otros arreglos; como se explica a continuación:

i) **Definición de matrices por vectores fila**
Una matriz puede verse como un vector columna cuyas componentes son vectores fila, así una matriz es un vector de tamaño $n \times 1$ cuyas componentes son vectores de tamaño $1 \times m$.

$$\underbrace{\begin{bmatrix} x_{11} & x_{12} & \cdots & x_{1m} \\ \vdots & \vdots & \ddots & \vdots \\ x_{i1} & x_{i2} & \cdots & x_{im} \\ \vdots & \vdots & \ddots & \vdots \\ x_{n1} & x_{n2} & \cdots & x_{nm} \end{bmatrix}}_{m} \quad n \left\{ \right. \quad = \quad \underbrace{\begin{bmatrix} \begin{bmatrix} x_{11} & x_{12} & \cdots & x_{1m} \end{bmatrix} \\ \vdots \\ \begin{bmatrix} x_{i1} & x_{i2} & \cdots & x_{im} \end{bmatrix} \\ \vdots \\ \begin{bmatrix} x_{n1} & x_{n2} & \cdots & x_{nm} \end{bmatrix} \end{bmatrix}}_{1} \quad \left. \right\} n$$

ii) **Definición de matrices por vectores columna**
Una matriz puede verse como un vector fila cuyas componentes son vectores columna, así una matriz es un vector de tamaño $1 \times m$ cuyas componentes son vectores de tamaño $n \times 1$.

$$\underbrace{\begin{bmatrix} x_{11} & \cdots & x_{1j} & \cdots & x_{1m} \\ x_{21} & \cdots & x_{2j} & \cdots & x_{2m} \\ \vdots & \ddots & \vdots & \ddots & \vdots \\ x_{n1} & \cdots & x_{nj} & \cdots & x_{nm} \end{bmatrix}}_{m} \quad n \left\{ \right. \quad = \quad \underbrace{\begin{bmatrix} \begin{bmatrix} x_{11} \\ x_{21} \\ \vdots \\ x_{n1} \end{bmatrix} & \cdots & \begin{bmatrix} x_{1j} \\ x_{2j} \\ \vdots \\ x_{nj} \end{bmatrix} & \cdots & \begin{bmatrix} x_{1m} \\ x_{2m} \\ \vdots \\ x_{nm} \end{bmatrix} \end{bmatrix}}_{m} \quad \left. \right\} 1$$

11.2.1. El conjunto de las matrices

En el capítulo 10 sobre arreglos, se definió un arreglo a partir del producto generalizado de un conjunto $\mathbb{T}$. El conjunto de los arreglos $\mathbb{T}^*$ se definió como la unión de todos los productos cartesianos del conjunto $\mathbb{T}$, de la siguiente manera

$$\mathbb{T}^* = \bigcup_{m\in\mathbb{N}} \mathbb{T}^m$$

A partir del concepto de arreglo y usando la definición i) de matrices por vectores fila se puede ahora definir el *conjunto de las matrices* $\mathbb{T}^{**}$ como la unión de todos los productos cartesianos del conjunto de los arreglos del conjunto $\mathbb{T}$, de la siguiente manera

$$\mathbb{T}^{**} = \bigcup_{n\in\mathbb{N}} \left(\bigcup_{m\in\mathbb{N}} \mathbb{T}^m \right)^n$$

El producto externo debe entenderse como un producto cartesiano que genera vectores columna y que internamente genera vectores fila, así como en la definición i).

Un elemento genérico del conjunto $\mathbb{T}^{**}$ es de la forma $(\mathbb{T}^m)^n$, donde n es el número de filas y m es el número de columnas. Para abreviar, de aquí en adelante se utilizará la notación

$$(\mathbb{T}^m)^n \Leftrightarrow \mathbb{T}^{n\times m}.$$

11.3. Las matrices en computación

Dado un conjunto de tipo de datos $\mathbb{T}$, a partir de la definición del conjunto de matrices se define el conjunto de las matrices o vectores bidimensionales $\mathbb{T}^{**}$ del tipo de datos $\mathbb{T}$. Así, el conjunto de las matrices del tipo de datos $\mathbb{T}$ es una colección de variables del tipo de datos $\mathbb{T}$ que están doblemente subindicadas, esto es, que se accede a ellas por medio de un par de indices que especifican una componente particular en una matriz, para esto es necesario saber el número de su fila (primer subíndice) y el de su columna (segundo subíndice).

Dada una matriz $\mathtt{X} \in \mathbb{T}^{**}$, para acceder en C++ a la variable almacenada en la componente (i, j) se utiliza la notación

$$\mathtt{X}_{ij} \equiv \mathtt{X[i][j]}$$

Nota: cuando se define una matriz de tamaño $n \times m$ en C++, es necesario tener en cuenta que la primera componente de la matriz está en la posición $(0, 0)$ en vez de $(1, 1)$ y la última componente estará ubicada en la posición $(n - 1, m - 1)$ en vez de (n, m), así como fue estudiada previamente.

11.3.1. Funciones para utilizar matrices

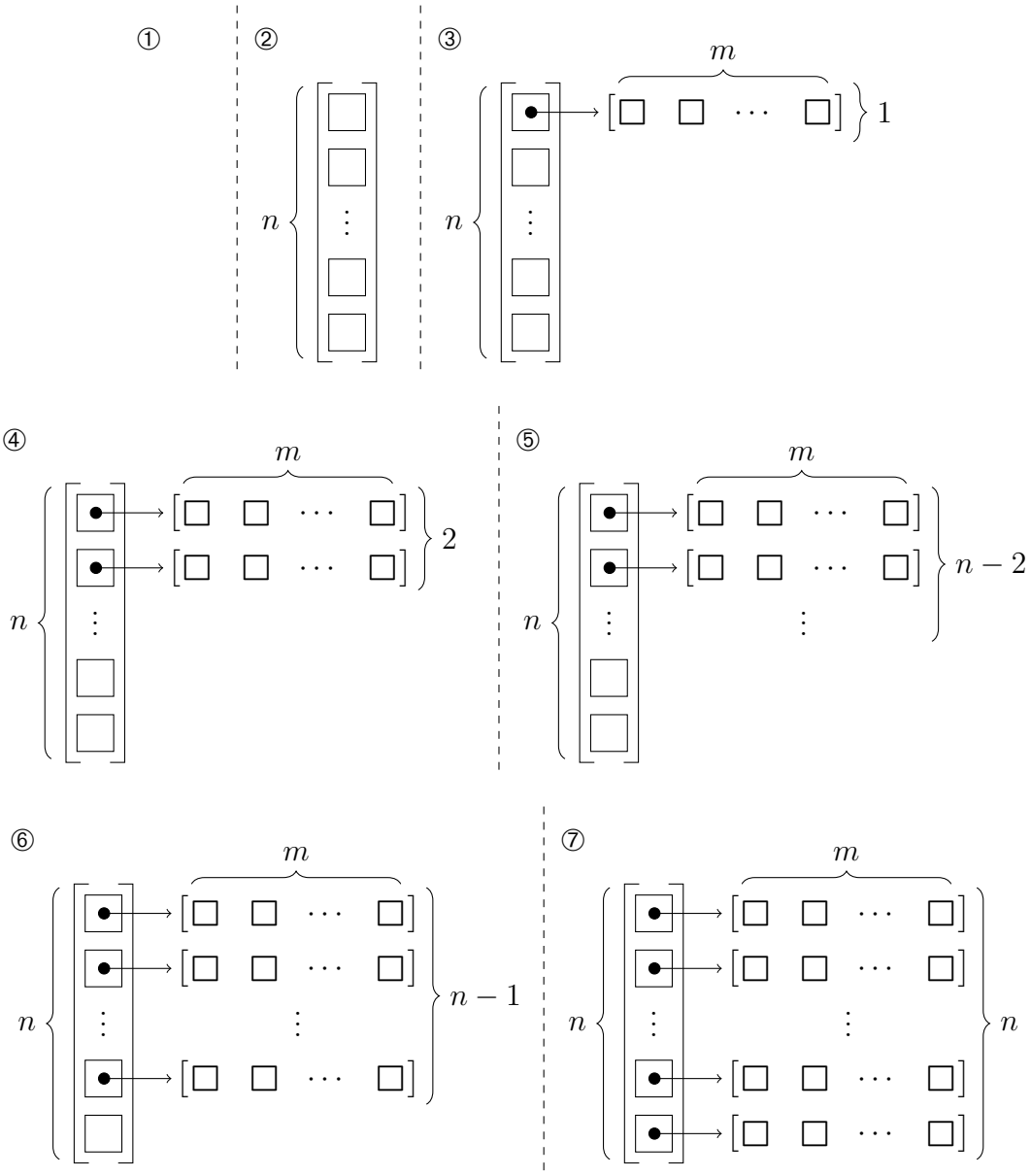
11.3.1.1. Creación de matrices

Matemáticamente se definirá una rutina para la creación de una matriz de tipo de datos $\mathbb{T}$ como aquella rutina que dado un $n \in \mathbb{N}$, que representa el número de componentes del arreglo columna, y $m \in \mathbb{N}$ que correspondiente al tamaño del cada arreglo fila, retornará una matriz de $\mathbb{T}^{**}$ de la siguiente manera

$$crear\_matriz\_T : \mathbb{N} \times \mathbb{N} \to \mathbb{T}^{**}$$
$$(n, m) \mapsto X, \quad \text{donde} \quad X \in \mathbb{T}^{n \times m} \subsetneq \mathbb{T}^{**}$$

En C++ se traduce

```
T** crear_matriz_T(int n, int m){
   T** X = new T*[n]; // define un arreglo columna de n componentes
   for(int i = 0; i < n; i++){
      X[i] = new T[m]; // crea y asigna cada arreglo fila de tamaño m
   };
   return X;
};
```



Ejemplo. Para crear una matriz de tipo entero se tiene la siguiente función

```
int** crear_matriz_int(int n, int m){
   int** X = new int*[n];
   for(int i = 0; i < n; i++){
      X[i] = new int[m];
   };
   return X;
};
```

11.3.1.2. Eliminación de matrices

Para eliminar una matriz, así como en el caso de los arreglos, se debe entender que lo que ocurre es que la porción del espacio de la memoria que se utiliza para almacenar los arreglos fila de la matriz se regresan al sistema operativo para que este disponga de esa memoria para almacenar nueva información.

Matemáticamente se puede modelar esto como una función que dada la matriz X, junto con el número de filas n y el número de columnas m (aunque este último no se requiere, pero se suele utilizar), se retorna la porción del espacio de la memoria $\mathcal{M}_X$ que es utilizado para almacenar la matriz X, esto se hace liberando cada uno de los arreglos fila que conforman la matriz, de esta manera la función se puede escribir así

$$\begin{aligned} liberar\_matriz\_T : \mathbb{T}^{**} \times \mathbb{N} \times \mathbb{N} &\to \wp(\mathcal{M}) \\ (X, n, m) &\mapsto \mathcal{M}_X \end{aligned}$$

Para traducir esta función a C++ se debe tener en cuenta que la función no retorna un valor de un tipo de datos, si no que se regresa memoria al sistema operativo, por lo que la función regresa un espacio que esta vacío y listo para utilizarse de nuevo.

El desbloqueo de la memoria y la liberación del espacio utilizado por cada uno de los arreglos fila X_i se escribe en C++ mediante la instrucción

```
delete[] X[i];
```

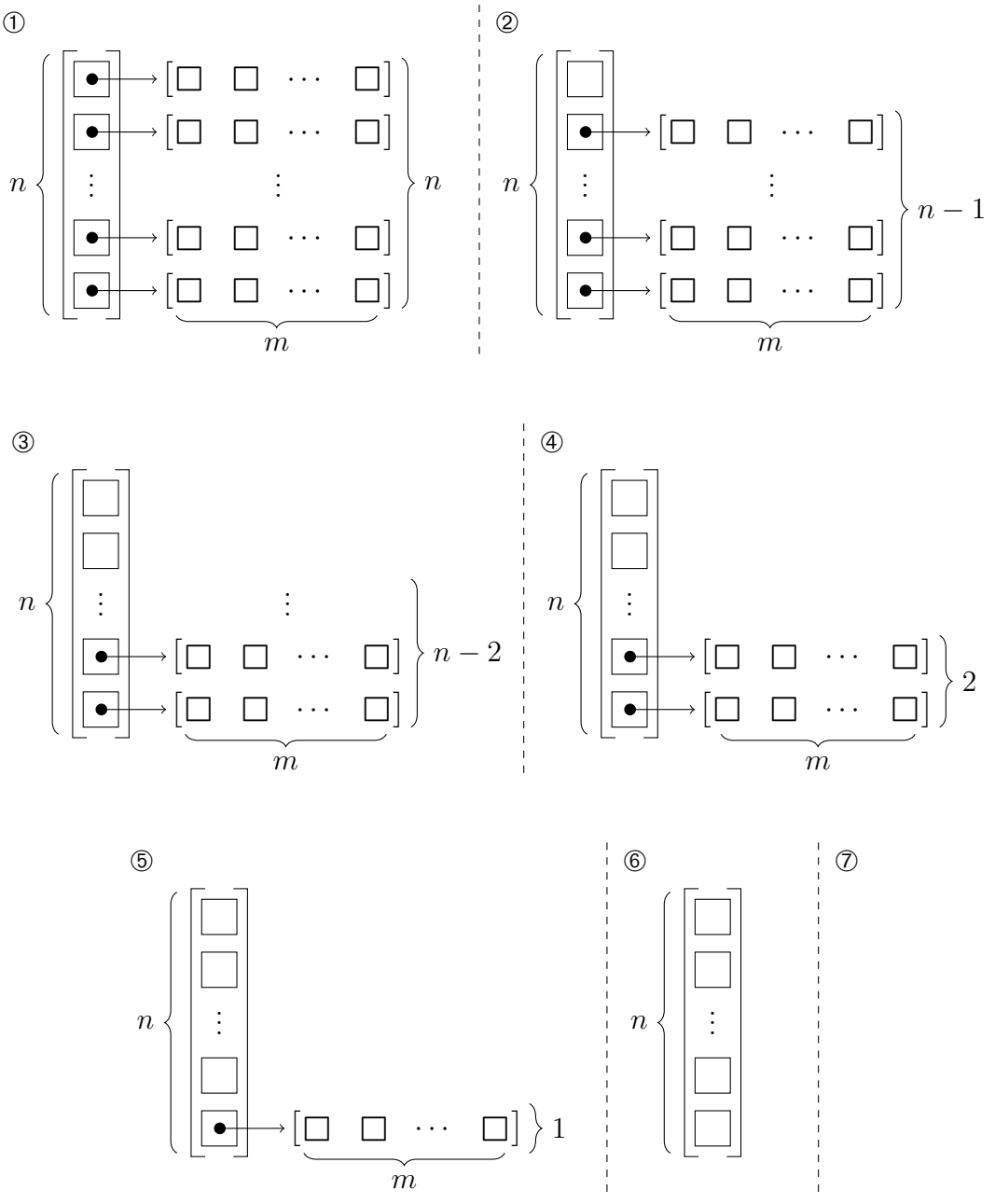
Luego de liberar cada uno de los arreglos fila se debe liberar el arreglo columna que los contenía, esto se escribe en C++ mediante la instrucción

```
delete[] X;
```

Al igual que en el caso de los arreglos, para decir que se retorna memoria vacía, en C++ esto se escribe como `void` y en el cuerpo de la función se utiliza la instrucción `return;` para indicar que se retorno una porción de espacio que esta vació y que queda listo para usarse de nuevo.

De esta manera, la traducción de la función en C++ se escribe de la siguiente manera

```
void liberar_matriz_T(T** X, int n, int m){
   for(int i = 0; i < n; i++){
      delete[] X[i]; // libera la memoria usada por los n arreglos fila
   };
   delete[] X; // libera la memoria usada por el arreglo columna
   return;
};
```



Ejemplo. Para liberar la memoria usada por una matriz de tipo entero se tiene la siguiente función

```
void liberar_matriz_int(int** X, int n, int m){
   for(int i = 0; i < n; i++){
      delete[] X[i];
   };
   delete[] X;
   return;
};
```

11.3.1.3. Matrices y flujos de datos

Dada una matriz de tipo $\mathbb{T}$, es posible realizar operaciones de lectura y escritura sobre flujos de datos, y dichas operaciones se realizan de la siguiente manera:

Lectura de matrices: para la entrada o lectura de una matriz desde un flujo de datos, se puede utilizar la siguiente función

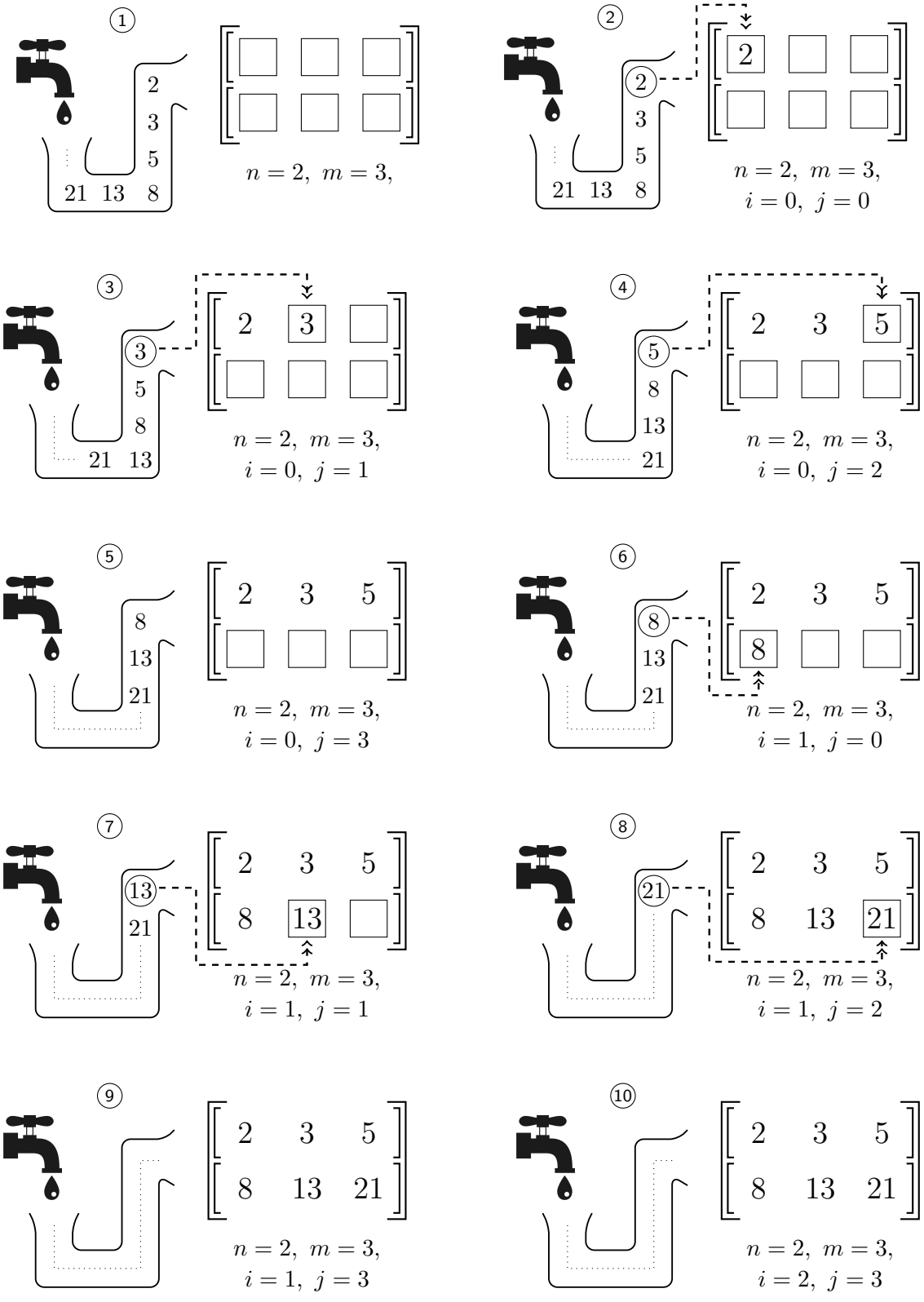
$$\begin{aligned} leer\_matriz\_T : \mathcal{IS} \times \mathbb{T}^{**} \times \mathbb{N} \times \mathbb{N} &\to \mathbb{T}^{**} \\ (is, X, n, m) &\mapsto X, \quad \text{donde} \qquad X_{ij} = leer\_T(is), \\ &\qquad\qquad\qquad\quad \forall i = 1, 2, 3, \ldots, n, \\ &\qquad\qquad\qquad\quad \forall j = 1, 2, 3, \ldots, m \end{aligned}$$

En C++ se traduce así (como se mencionó previamente, las matrices en C++ comienzan en la posición $(0, 0)$)

```
T** leer_matriz_T(istream& is, T** X, int n, int m){
   for(int i = 0; i < n; i++){
      for(int j = 0; j < m; j++){
         is >> X[i][j];
      };
   };
   return X;
};
```

Ejemplo. En C++ para leer una matriz de tipo entero se utiliza la siguiente función

```
int** leer_matriz_int(istream& is, int** X, int n, int m){
   for(int i = 0; i < n; i++){
      for(int j = 0; j < m; j++){
         is >> X[i][j];
      };
   };
   return X;
};
```



Escritura de matrices: para enviar o escribir una matriz en un flujo de datos, se puede definir la siguiente función

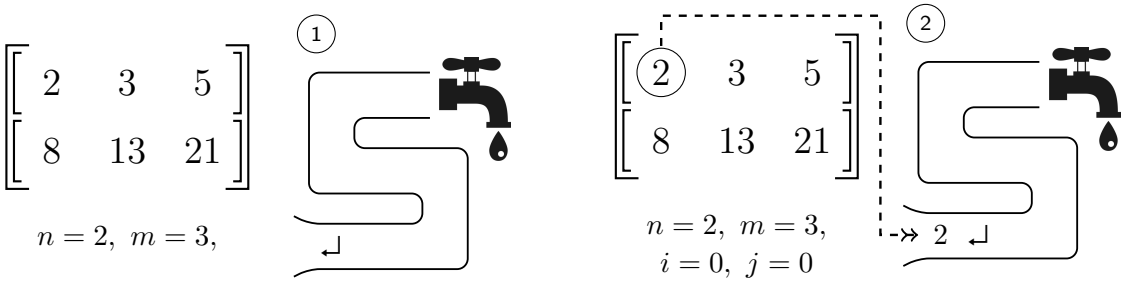
$$escribir\_matriz\_T : \mathcal{OS} \times \mathbb{T}^{**} \times \mathbb{N} \times \mathbb{N} \to \mathcal{OS}$$
$$(os, X, n, m) \mapsto os, \quad \text{donde} \quad escribir\_T(X_{ij}, os),$$
$$\forall i = 1, 2, 3, \ldots, n,$$
$$\forall j = 1, 2, 3, \ldots, m$$

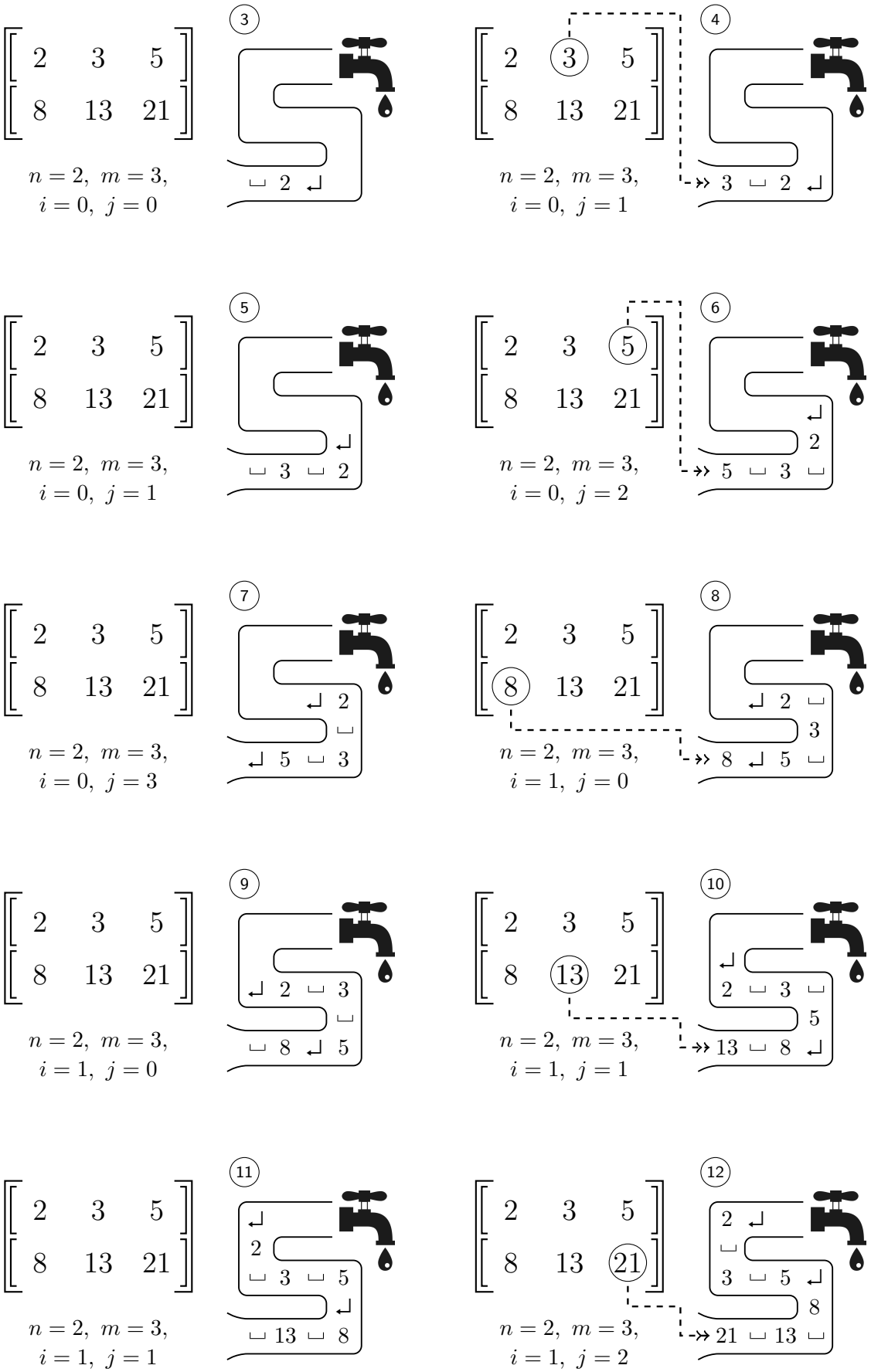
En C++ se traduce así

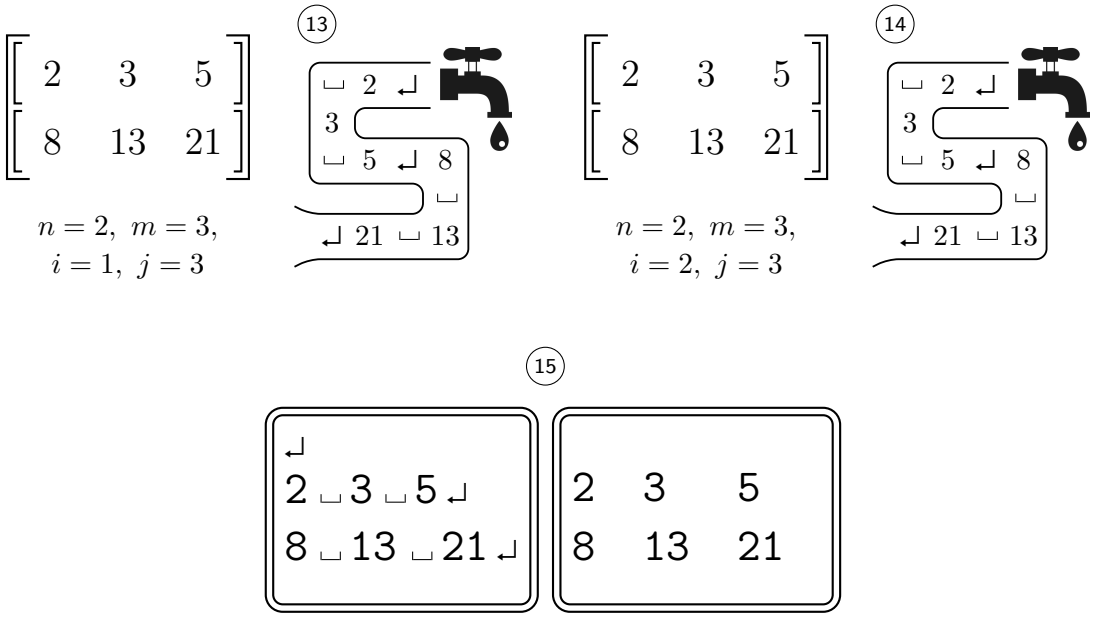
```
ostream& escribir_matriz_T(ostream& os, T** X, int n, int m){
   os << "\n";
   for(int i = 0; i < n; i++){
      for(int j = 0; j < m; j++){
         os << X[i][j];
         if(j < m - 1){
            os << "\t";
         };
      };
      os << "\n";
   };
   return os;
};
```

Ejemplo. En C++ para escribir una matriz de tipo entero se utiliza la siguiente función

```
ostream& escribir_matriz_int(ostream& os, int** X, int n, int m){
   os << "\n";
   for(int i = 0; i < n; i++){
      for(int j = 0; j < m; j++){
         os << X[i][j];
         if(j < m - 1){
            os << "\t";
         };
      };
      os << "\n";
   };
   return os;
};
```




3
2 3 5
8 13 21
n = 2, m = 3,
i = 0, j = 0
␣ 2 ↵
4
2 3 5
8 13 21
n = 2, m = 3,
i = 0, j = 1
↠ 3 ␣ 2 ↵
5
2 3 5
8 13 21
n = 2, m = 3,
i = 0, j = 2
␣ 3 ␣ 2 ↵
6
2 3 5
8 13 21
n = 2, m = 3,
i = 0, j = 3
↠ 5 ␣ 3 ␣ ↵ 2
7
2 3 5
8 13 21
n = 2, m = 3,
i = 1, j = 0
↵ 5 ␣ 3 ␣ ↵ 2 ␣
8
2 3 5
8 13 21
n = 2, m = 3,
i = 1, j = 1
↠ 8 ↵ 5 ␣ ↵ 2 ␣ 3
9
2 3 5
8 13 21
n = 2, m = 3,
i = 1, j = 2
␣ 8 ↵ 5 ␣ ↵ 2 ␣ 3 ␣
10
2 3 5
8 13 21
n = 2, m = 3,
i = 2, j = 0
↠ 13 ␣ 8 ↵ 2 ␣ 3 ␣ 5
11
2 3 5
8 13 21
n = 2, m = 3,
i = 2, j = 1
␣ 13 ␣ 8 ↵ 2 ␣ 3 ␣ 5 ↵
12
2 3 5
8 13 21
n = 2, m = 3,
i = 2, j = 2
↠ 21 ␣ 13 ␣ ↵ 2 ␣ 3 ␣ 5 ↵ 8



11.3.2. Ejemplos de funciones con matrices

Es posible utilizar lo visto en funciones para realizar diversidad de cálculos que involucren matrices.

Ejemplo. *El cuadrado de las componentes de matrices numéricas enteras*

Suponga que un archivo contiene unos datos numéricos enteros tales que consta de 5 líneas de texto y en cada línea hay dos números escritos que se encuentran separados por una tabulación así como se muestra a continuación

```
0	1
2	3
4	5
6	7
8	9
```

Una función general que permite construir una nueva matriz que contiene el cuadrado de cada componente de una matriz dada es

$$\begin{aligned} cuadrado\_matriz : \mathbb{Z}^{**} \times \mathbb{N} \times \mathbb{N} &\to \mathbb{Z}^{**} \\ (X, n, m) &\mapsto Y, \quad \text{donde} \quad Y_{ij} = X_{ij}^2, \\ &\qquad\qquad\qquad\quad \forall i = 1, 2, \ldots, n, \\ &\qquad\qquad\qquad\quad \forall j = 1, 2, \ldots, m \end{aligned}$$

Un programa completo en C++ que permite calcular el cuadrado de las componentes de una matriz obtenida a partir del archivo presentado anteriormente es

```
#include<iostream>
#include<cstdlib>
#include<fstream>

using  namespace  std;
```

```
int** crear_matriz_int(int n, int m){
   int** X = new int*[n];
   for(int i = 0; i < n; i++){
     X[i] = new int[m];
   };
   return X;
};
```

```
void liberar_matriz_int(int** X, int n, int m){
   for(int i = 0; i < n; i++){
      delete[] X[i];
   };
   delete[] X;
   return;
};
```

```
int** leer_matriz_int(istream& is, int** X, int n, int m){
   for(int i = 0; i < n; i++){
      for(int j = 0; j < m; j++){
         is >> X[i][j];
      };
   };
   return X;
};
```

```
ostream& escribir_matriz_int(ostream& os, int** X, int n, int m){
   os << "\n";
   for(int i = 0; i < n; i++){
      for(int j = 0; j < m; j++){
         os << X[i][j];
         if(j < m - 1){
            os << "\t";
         };
      };
      os << "\n";
   };
   return os;
};
```

```
int** cuadrado_matriz(int** X, int n, int m){
   int** Y = crear_matriz_int(n, m);
   for(int i = 0; i < n; i++){
      for(int j = 0; j < m; j++){
         Y[i][j] = X[i][j] * X[i][j];
      };
   };
   return Y;
};
```

```
int  main(){
   int n = 5;
   int m = 2;
   ifstream ifs("matriz_numeros.txt");
   ofstream ofs("matriz_cuadrados.txt");
   int** X = crear_matriz_int(n, m);
   X = leer_matriz_int(ifs, X, n, m);
   int** Y = cuadrado_matriz(X, n, m);
   escribir_matriz_int(ofs, Y, n, m);
   liberar_matriz_int(X, n, m);
   liberar_matriz_int(Y, n, m);
   ifs.close();
   ofs.close();
   cout  << "El calculo de la matriz fue exitoso\n";
   system("pause");
   return  EXIT_SUCCESS;
};
```

Ejemplo. *Tablas de multiplicar*
Se puede generar una matriz que represente las tablas de multiplicar multiplicando cada posición (i, j) de la matriz y almacenándola en la matriz:

$$\begin{aligned} tablas\_mult : \mathbb{Z}^{**} \times \mathbb{N} \times \mathbb{N} &\to \mathbb{Z}^{**} \\ (X, n, m) &\mapsto X, \quad \text{donde} \quad X_{ij} = i * j, \\ &\qquad\qquad \forall i = 1, 2, \ldots, n, \\ &\qquad\qquad \forall j = 1, 2, \ldots, m \end{aligned}$$

En C++:

```
int** tablas_mult(int** X, int n, int m){
   for(int i = 1; i <= n; i++){
      for(int j = 1; j <= m; j++){
         X[i-1][j-1] = i * j;
      };
   };
   return X;
};
```

Ejemplo. *Determinantes* 2×2
Dado un sistema de ecuaciones de dos ecuaciones con dos incógnitas

$$ax + by = c$$
$$a'x + b'y = c'$$

este sistema de ecuaciones se puede expresar mediante matrices de la siguiente manera

$$\begin{bmatrix} a & b \\ a' & b' \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} c \\ c' \end{bmatrix}$$

una solución del sistema es una pareja (x_0, y_0) tal que

$$ax_0 + by_0 = c \qquad \text{y} \qquad a'x_0 + b'y_0 = c'.$$

El sistema tiene una solución única si la expresión $a * b' - a' * b \neq 0$, a dicha expresión se le conoce como el determinante de la matriz, y efectivamente sirve para determinar si un sistema tiene una única solución, o si no tiene, o si no es única. A continuación se presenta una función que permite calcular el determinante de una matriz 2×2.

$$\begin{aligned} det\_2\_2 : \mathbb{R}^{2\times 2} &\to \mathbb{R} \\ X &\mapsto X_{11} * X_{22} - X_{21} * X_{12} \end{aligned}$$

En C++:

```
double det_2_2(int** X){
   return X[0][0]*X[1][1]-X[1][0]*X[0][1];
};
```

Ejemplo. *Producto escalar*

Una de los operaciones básicas sobre las matrices es el producto escalar. En éste se elige un valor sobre un campo (por ejemplo los reales $\mathbb{R}$) y se multiplica este valor por cada componente de la matriz.

Así, si $\alpha \in \mathbb{R}$ y $X = \begin{bmatrix} x_{11} & x_{12} & \cdots & x_{1m} \\ x_{21} & x_{22} & \cdots & x_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ x_{n1} & x_{n2} & \cdots & x_{nm} \end{bmatrix}$

entonces

$$Y = \alpha * X = \alpha * \begin{bmatrix} x_{11} & x_{12} & \cdots & x_{1m} \\ x_{21} & x_{22} & \cdots & x_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ x_{n1} & x_{n2} & \cdots & x_{nm} \end{bmatrix} = \begin{bmatrix} \alpha \cdot x_{11} & \alpha \cdot x_{12} & \cdots & \alpha \cdot x_{1m} \\ \alpha \cdot x_{21} & \alpha \cdot x_{22} & \cdots & \alpha \cdot x_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ \alpha \cdot x_{n1} & \alpha \cdot x_{n2} & \cdots & \alpha \cdot x_{nm} \end{bmatrix}$$

La definición de esta función sería formalmente

$$\begin{aligned} \mathit{producto\_escalar} : \mathbb{R} \times \mathbb{R}^{**} \times \mathbb{N} \times \mathbb{N} &\to \mathbb{R}^{**} \\ (\alpha, X, n, m) &\mapsto Y, \quad \text{donde} \quad Y_{ij} = \alpha * X_{ij}, \\ & \qquad\qquad\quad \forall i = 1, 2, \ldots, n, \\ & \qquad\qquad\quad \forall j = 1, 2, \ldots, m \end{aligned}$$

En C++:

```
double** producto_escalar(double alpha, double** X, int n, int m){
   double** Y = crear_matriz_double(n,m);
   for(int i = 0; i < n; i++){
      for(int j = 0; j < m; j++){
         Y[i][j] = alpha*X[i][j];
      };
   };
   return X;
};
```

Ejercicios

1. Desarrollar un algoritmo que permita hallar el vector solución de un sistema de dos ecuaciones con dos incógnitas, utilizando la regla de Cramer.

2. Desarrollar un algoritmo que permita hallar la matriz inversa de una matriz 2×2, si ésta existe.

3. Desarrollar un algoritmo que permita calcular el determinante de una matriz 3×3, utilizando la regla de Sarrus.

4. Desarrollar un algoritmo que permita hallar el vector solución de un sistema de tres ecuaciones con tres incógnitas, utilizando la regla de Cramer.

5. Desarrollar un algoritmo que permita calcular la traza de una matriz cuadrada.

6. Desarrollar un programa que sume los elementos de una columna dada de una matriz.

7. Desarrollar un programa que sume los elementos de una fila dada de una matriz.

8. Desarrollar un algoritmo que permita sumar dos matrices de números reales.

9. Desarrollar un algoritmo que permita hallar la transpuesta de una matriz.

10. Desarrollar un algoritmo que permita multiplicar dos matrices de números reales.

11. Desarrollar un algoritmo que determine si una matriz es mágica. Se dice que una matriz cuadrada es mágica si la suma de cada una de sus filas, de cada una de sus columnas y de cada diagonal es igual. Ejemplo:

$$\begin{bmatrix} 8 & 1 & 6 \\ 3 & 5 & 7 \\ 4 & 9 & 2 \end{bmatrix}$$

12. Desarrollar un algoritmo que dado un entero, reemplace en una matriz todos los números mayores al número dado por un uno y todos los menores o iguales por un cero.

    Si el número dado es: 5 y una matriz en el arreglo es:

$$\begin{bmatrix} 8 & 1 & 6 \\ 3 & 5 & 7 \\ 4 & 9 & 2 \end{bmatrix}$$

    La matriz de salida es:

$$\begin{bmatrix} 1 & 0 & 1 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

13. Desarrollar un programa que genere una matriz marco cuadrada de tamaño $n \times n$.

    Entrada: $n = 3$

$$\begin{bmatrix} 1 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

14. Desarrollar un programa que tome un arreglo de tamaño n^2 y llene en espiral hacia adentro una matriz cuadrada de tamaño n.

    Ejemplo: arreglo de entrada: $[1, 2, 3, 4, 5, 6, 7, 8, 9]$, la matriz de salida es:

$$\begin{bmatrix} 1 & 2 & 3 \\ 8 & 9 & 4 \\ 7 & 6 & 5 \end{bmatrix}$$

15. Desarrollar un algoritmo que permita resolver un sistema de n ecuaciones con n incógnitas, usando el método de eliminación de Gauss-Jordan.

16. Desarrollar un algoritmo que permita hallar la matriz inversa de una matriz cuadrada, si ésta existe.

17. Desarrollar un algoritmo que permita calcular el determinante de una matriz cuadrada, usando el teorema de Laplace (desarrollo por menores y cofactores).

Capítulo 12

Cadenas de carácteres

Hasta ahora se ha trabajado con tipos de datos primitivos y arreglos. Entre los tipos de datos primitivos se encuentran los caracteres que se definieron matemáticamente como $ASCII$.

Los caracteres $ASCII$ (American Standard Code for Information Interchange) permiten representar letras, numeros y símbolos. Inicialmente se definieron de 7 bits (128 caracteres) y luego se extendieron a 8 bits (256 caracteres). Decir que $x \in ASCII$, en C++ equivale a decir `char x`.

Sobre los caracteres se pueden tener las siguientes operaciones:

```
char letra = 'A';  //Asigna el caracter 'A'
char letra2 = 65; // Asigna la letra A
                  //a letra 2, pues A
                  // es el caracter en decimal
                  // 65 del ASCII
cout << letra + letra2  //imprime el numero 130
cout << (char)(letra + letra2) //imprime e tildada
if(letra == letra2){ //Se pueden hacer
  cout << "igual";   //comparaciones entre char
}
cout << '0' + '1' //imprime 97
cout << (char)('0' + '1') //imprime 'a'
```

Como se vió en la parte de flujos, existen tambien caracteres especiales como los siguientes:

```
cout << (int)(' '); //imprime 32 que corresponde al caracter
          // espacio
cout << (int)('\n'); // imprime el numero 13
// que en ASCII es el caracter
//new line
cout << (int)('\0'); //imprime el valor cero que
         //equivale a NUL o fin de cadena
```

Las cadenas de caracteres corresponden a arreglos de char ($ASCII*$) y toda cadena termina con el caracter numero cero o NUL que representa fin de linea.

Es posible definir cadenas de forma estática, si se definen así no podran modificarse o reasignarse:

```
char* mensaje = "hola mundo\n\0"; //creacion
                                  // de cadena
                                  // estatica
```

A continuación hay un ejemplo que muestra como cadenas pueden ser vistas como arreglos. Carga la cadena mensaje con el texto "hola.[ei]mprime un cambio de linea. Nótese que el tamaño del arreglo es de 100, pero la cadena resultante es de tamaño 5 sin incluir el caracter de terminacion de cadena.

```
int n = 100;
char* mensaje = new cadena[n];
mensaje[0] = 'h';
mensaje[1] = 'o';
mensaje[2] = 'l';
mensaje[3] = 'a';
mensaje[4] = '\n';
mensaje[5] = '\0';
cout << mensaje;
```

Sobre arreglos definimos algunas funciones clave que también aplican para cadenas pero con algunos cambios que veremos a continuación:

Funciones constructoras: Las funciones constructoras permiten como su nombre lo indica construir una cadena, cuando vimos arreglos la definición general era esta:

$$crear\_arreglo : \mathbb{N} \longrightarrow A*$$
$$(n) \longmapsto x|x \in A^n \subseteq A^*$$

En C++ se traduce:

```
A* crear_arreglo(int n){
return new A[n];
};
```

Con cadenas pasa lo mismo:

$$crear\_cadena : \mathbb{N} \longrightarrow ASCII*$$
$$(n) \longmapsto x|x \in ASCII^n \subseteq ASCII^*$$

En C++ se traduce:

```
char* crear_cadena(int n){
return new char[n];
};
```

Funciones Liberadoras: Estas funciones permiten devolver la memoria empleada por un TDA al sistema operativo. Los tipos de datos primitivos no requieren esta definición pero tipos de datos como los arreglos o los TDA escritos por un programador deben ser liberados. Están asociadas al operador `delete` en C++. Para arreglos se tenía:

$$liberar\_arreglo : A^* \times \mathbb{N} \longrightarrow \varnothing$$
$$(x, n) \longmapsto$$

En C++ se traduce:

```
void liberar_arreglo(A* x, int n){
delete[] x;
};
```

Para las cadenas se tiene lo mismo, pero sin tener el n de tamaño.

$$liberar\_cadena : ASCII^* \longrightarrow \varnothing$$
$$(x, n) \longmapsto$$

En C++ se traduce:

```
void liberar_arreglo(char* x){
delete[] x;
};
```

Funciones de persistencia: Acá esta la diferencia con arreglos. No se le preguntará al usuario cuantos caracteres tiene el nombre ni leerlos letra por letra, nos toca aproximar un n que en este caso tambien lo enviamos como parametro (vease el main mas adelante). Lo interesante de las cadenas con respecto a los arreglos de enteros y reales es que podemos leer toda la cadena con una sola instrucción de la librería `iostream`:

$$leer\_cadena : \mathbb{I}S \times ASCII^* \times \mathbb{N} \longrightarrow ASCII^*$$
$$(is, x, n) \longmapsto x | x = leer\_cadena(is, n)$$

```
char* leer_cadena(istream& is, char* x, int n){
is.getline(x, n);
};
```

En este caso el n puede ser aproximado por el programador, obsérvese que no se empleó `is >> x` pues no podría leer nombres por ejemplo "Sandra Milena"quedaría "Sandra", getline leerá el flujo hasta encontrar un salto de linea o cumplirse la longitud de cadena dada.

Para imprimir una cadena si podemos utilizar el operador `<<`:

$$escribir\_cadena : \mathbb{O}S \times ASCII^* \longrightarrow \mathbb{O}S$$
$$(os, x) \longmapsto os|escribir\_cadena(os, x)$$

En C++:

```
ostream& escribir_cadena(ostream& ofs, char* x){
   ofs << x << "\t";
   return ofs;
};
```

A diferencia de los arreglos no es necesario imprimir elemento a elemento con un ciclo o una rutina recursiva.

Otras funciones: Como se suele operar en cadenas con los datos que proporciona un usuario es necesario tener una función especial que nos indique cuantos caracteres hay al final de una cadena, esto es contar cuantos caracteres hay antes del caracter '\0'.

Recursivamente esta función puede definirse como obtener la longitud de una cadena:

$$longitud\_cadena\_parcial : ASCII^* \times \mathbb{N} \longrightarrow \mathbb{N}$$
$$(x, i) \longmapsto 0 \ si \ x[i] =' \backslash 0'$$
$$\longmapsto 1 + longitud\_cadena\_parcial(x, i+1) \ en \ otro \ caso$$

$$longitud\_cadena : ASCII^* \longrightarrow \mathbb{N}$$
$$(x) \longmapsto longitud\_cadena\_parcial(x, 0)$$

En C++ se traduce:

```
int longitud_cadena_p(char* str, int i){
   if(str[i] == '\0'){
      return 0;
   };
   return 1 + longitud_cadena_p(str, i+1);
};

int longitud_cadena(char* str){
   return longitud_cadena_p(str, 0);
};
```

Con esta función de longitud se pueden definir funciones interesantes como copiar cadena:

$copiar\_cadena : ASCII^* \longrightarrow \mathbb{ASCII}$
$(str) \longmapsto strcp | strcp = crear\_cadena(longitud\_cadena(str) + 1), strcp_i = str_i \ \forall_{i=0,1,2,..,longitud\_cadena(str)}, strcp_{(longitud\_cadena(str))} =' \backslash 0'$

```
char* copiar_cadena(char* str){
   int i;
   int lstr = longitud_cadena(str);
   char* strcp = crear_cadena(lstr+1);
   for(i=0; i < lstr; i++){
      strcp[i] = str[i];
   };
   strcp[lstr] = '\0';
   return strcp;
};
```

12.1. Codigo Completo

El código completo de cadenas quedaría así:

```
#include <iostream>

using namespace std;

int longitud_cadena_p(char* str, int i){
   if(str[i] == '\0'){
      return 0;
   };
   return 1 + longitud_cadena_p(str, i+1);
};

int longitud_cadena(char* str){
   return longitud_cadena_p(str, 0);
};

char* crear_cadena(int n){
   return new char[n];
};

char* copiar_cadena(char* str){
   int i;
   int lstr = longitud_cadena(str);
   char* strcp = crear_cadena(lstr+1);
   for(i=0; i < lstr; i++){
      strcp[i] = str[i];
   };
   strcp[lstr] = '\0';
   return strcp;
};

void liberar_cadena(char* str){
   delete[] str;
};

char* leer_cadena(istream& is, char* str, int n){
   is.getline(str, n);
   return str;
};

ostream& escribir_cadena(ostream& os, char* str){
   os << str << "\n";
};

int main(){
   int n = 100;
   /* Crea una cadena de tamano 100 */
   char* str = crear_cadena(n);
   char* str_copia = crear_cadena(n);

   /* Lee la cadena de la consola */
```

Ejercicios

- Elabore un programa que dada una letra cuente cuantas ocurrencias de esta letra hay.
- Elabore un programa que dada una cadena diga si todos los simbolos de la cadena son letras.
- Elabore un programa que dada una cadena cuente las consonantes en dicha cadena.
- Desarrollar un algoritmo que invierta una cadena de caracteres (la cadena invertida debe quedar guardada en una variable aparte)
- Desarrollar un algoritmo que determine si una cadena de caracteres es palíndrome. Una cadena se dice palíndrome si al invertirla es igual a ella misma. Ejemplos:
  - .ªla.es palindrome
  - .ªnita lava la tina"No es palindrome, pues al invertirla con espacios no es exactamente igual a la original.
  - "los estudiantes de programación leyeron toda la guía"no es palindrome.
  - robas ese sabor.es palindrome
- Desarrollar un algoritmo que realice el corrimiento circular a izquierda de una cadena de caracteres. El corrimiento circular a izquierda es pasar el primer carácter de una cadena como ultimo caracter de la misma. Ejemplo: "Los estudiantes hicieron bien el taller"quedaría .ºs estudiantes hicieron bien el tallerL"
- Desarrollar un algoritmo que reciba como entrada dos cadenas y determine si la primera es subcadena de la segunda. (No se deben usar operaciones de subcadenas propias del lenguaje de programación). Ejemplos: La cadena "prosa.es subcadena de la cadena "la prosa debe ser armoniosa"La cadena "pepito"no es subcadena de la cadena .el torpe pito de aire". La cadena "pe pito"si esta incluida en la cadena .el torpe pito de aire"

Capítulo 13

Tipos de datos abstractos (TDA)

Hasta ahora se ha trabajado con tipos de datos primitivos y arreglos. Los Tipos de Datos Abstractos (TDA) nos permiten realizar una mayor abstracción de las cosas que existen en el mundo real, pues nos permiten olvidar detalles sobre implementaciones y mantener la esencia.

Cuando se tiene un tipo de dato primitivo como los enteros, se tiene la definición de entero $\mathbb{Z}$, que representa datos de tipo entero. En los enteros, se tienen definidas las operaciones aritméticas $+, *, -, /, mod$ que nos permiten realizar operaciones con este tipo de datos sin conocer como se esta realizando una suma o una división a nivel de máquina.

En este libro se definieron funciones de lectura y escritura a flujos de datos para enteros. Una vez definida la función si otra persona desea utilizarla lo unico que debe hacer es llamar a la función e ignorar los detalles implementacion de la función en si.

13.1. Estructura y Operaciones

Se puede considerar un TDA como una estructura que tiene una serie de operaciones. Como estructura, los tipos abstractos de datos permiten agrupar varios datos que mantengan algún tipo de relación aunque sean de distinto tipo. Si se tienen los conjuntos $A_1, A_2, A_3, ..., A_n$, que definen diferentes tipos de datos se puede definir el TDA A dado el producto cartesiano como $A = A_1 \times A_2 \times A_3 \times ... \times A_n$.

Las operaciones que se pueden definir sobre un tipo de datos son las constructoras, las analizadoras, las modificadoras, las liberadoras, y otras funciones de utilidad que se requieran en un TDA.

Funciones constructoras: Las funciones constructoras permiten como su nombre lo indica construir la estructura del TDA dado. Debe tenerse en cuenta que las variables no estan definidas. Para el caso de los enteros si se desea definir una variable como entera $x \in \mathbb{Z}$ la equivalencia de dicha expresión en un lenguaje de programación como C++ sería `int x`. Si se tratase de un arreglo de tipo entero $x \in \mathbb{Z}^*$, su definición en C++ sería la encargada de crear la variable como `int* x`. Los tipos primitivos de datos no necesitan ser construidos en memoria pero en el caso de TDA dinámicos definidos por un programador en lenguaje C++ se requiere una asignación en la memoria con el operador `new`.

Funciones Modificadoras: Estas funciones permiten cambiar un dato de un TDA, están relacionadas con las asignaciones.

Funciones Analizadoras: Estas funciones permiten obtener un dato miembro de un TDA.

Funciones Liberadoras: Estas funciones permiten devolver la memoria empleada por un TDA al sistema operativo. Los tipos de datos primitivos no requieren esta definición pero tipos de datos como los arreglos o los TDA escritos por un programador deben ser liberados. Están asociadas al operador `delete` en C++.

Funciones de persistencia: Son funciones que permiten mantener el dato a través del tiempo. En este caso se han utilizado los flujos de datos para leer o escribir los datos en archivos o enviarlos a la salida estándar de la consola.

13.2. Definición de un TDA

Para definir un TDA, llamado A partiendo de la definición de producto cartesiano $A = A_1 \times A_2 \times A_3 ... \times A_n$ La definición de dicho TDA en C++ será la siguiente:

```
struct base_A{
   A_1 a_1;
   A_2 a_2;
   A_3 a_3;
   .
   .
   .
   A_n a_n;
};

typedef base_A* A;
```

La palabra `struct` declara la construcción de la estructura como el producto cartesiano y `typedef` nos permite nombrar un puntero a la estructura base_A y así poder manipular el TDA como parámetro en una función o retornarlo como salida de funciones sin realizar cambios adicionales utilizando memoria dinámica.

Como ejemplo pueden definirse los numeros complejos $\mathbb{C}$ como un TDA constituido por dos numeros reales, uno que representa la parte real del y otro que representa la parte imaginaria. $\mathbb{C} = \mathbb{R} \times \mathbb{R}$. En C++ se declararan así:

```
struct base_complejo{
  double x; //parte real
  double y; //parte imaginaria
};

typedef base_complejo* complejo;
```

13.2.1. Funciones Constructoras

Para construir un TDA se define la operación crear, que va matemáticamente de la declaración del producto cartesiano y retornará la estructura. En general se tiene:

$crear\_A : A_1 \times A_2 \times A_3 \times ... \times A_n \longrightarrow A$
$(a_1, a_2, a_3, ..., a_n) \longmapsto (a_1, a_2, a_3, ..., a_n)$

En C++:

```
A crear_A(A_1 a_1, A_2 a_2, A_3 a_3, ... , A_n a_n){
    A a = new base_A;
    a->a_1 = a_1;
    a->a_2 = a_2;
    a->a_3 = a_3;
    .
    .
    .
    a->a_n = a_n;
    return a;
};
```

Si $A_i \forall_{i=1,2,3,...,n}$, es un tipo de datos no primitivo se aprovecha esta definición para llamar la función creadora del tipo A_i dado con el parametro x_i respectivo.

Para el caso de los números complejos, la función creadora de complejo esta definida matemáticamente como:

$crear\_complejo : \mathbb{R} \times \mathbb{R} \longrightarrow \mathbb{C}$
$(x, y) \longmapsto (x, y)$

En C++:

```
complejo crear_complejo(double x, double y){
   complejo z = new base_complejo;
   z->x = x; //parte real
   z->y = y; //parte imaginaria
   return z;
};
```

Para obtener un elemento miembro de la estructura se emplea el operador `->`.

13.2.2. Funciones Analizadoras

Las funciones analizadoras permiten obtener un miembro de una estructura, si se define la estructura de un TDA como un producto cartesiano, la función analizadora devolverá la proyección i-esima (notada π_i). Matemáticamente las funciones analizadoras se definen así:

$\pi_i : A \longrightarrow A_i$
$(a_1, a_2, a_3, ..., a_n) \longmapsto a_i$

En C++:

```
A_i parte_i(A a){
   return a->a_i;
};
```

Una función analizadora para un numero complejo es la obtención de la parte real del numero:

$\pi_1 : \mathbb{C} \longrightarrow \mathbb{R}$
$(x, y) \longmapsto x$

Para la parte imaginaria del número:

$\pi_2 : \mathbb{C} \longrightarrow \mathbb{R}$
$(x, y) \longmapsto y$

En C++:

```
double parte_real(complejo z){
   return z->x;
};
```

```
double parte_imaginaria(complejo z){
   return z->y;
};
```

13.2.3. Funciones Modificadoras

Están asociadas a las asignaciones de un tipo de dato que hace parte de la estructura del TDA y su función es inyectar un dato en la estructura reemplazando el valor de un determinado tipo de dato.

En general se tiene la inyección i_k del miembro de la estructura k como:
$i_k : A \times A_k \longrightarrow A$
$((a_1, a_2, a_3, ..., a_n), b) \longmapsto c | c_j = a_j \forall_{j=1,2,3,...,n \wedge j \neq k} \wedge c_k = b$

Para el TDA complejo se tienen dos inyecciones, uno para la parte real:

$i_1 : \mathbb{C} \times \mathbb{R} \longrightarrow \mathbb{C}$
$((x, y), r) \longmapsto (r, y)$

Y otra para la parte imaginaria:
$i_2 : \mathbb{C} \times \mathbb{R} \longrightarrow \mathbb{C}$
$((x, y), r) \longmapsto (x, r)$

En C++:

```
complejo modificar_parte_real(complejo z, double x){
   z->x = x;
   return z;
};

complejo modificar_parte_imaginaria(complejo z, double y){
   z->y = y;
   return z;
};
```

13.3. Otras Funciones y Funciones Analizadoras

Constituyen cualquier función que se requiera sobre el tipo de dato dado. En el caso de los numeros complejos la suma de dos números complejos se define como otro numero complejo cuya parte real es la suma de las partes reales y cuya parte imaginaria es la suma de las partes imaginarias.

$sumar\_complejo : \mathbb{C} \times \mathbb{C} \longrightarrow \mathbb{C}$
$((x, y), (v, w)) \longmapsto (x + v, y + w)$

En C++ se tiene:

```
complejo sumar_complejo(complejo z1, complejo z2){
   return crear_complejo(z1->x + z2->x, z1->y + z2->y);
};
```

13.4. Funciones de Persistencia o E/S

Las funciones de persistencia o de entrada y salida estan basadas en los conceptos de flujos de datos.

Para leer un TDA de un flujo de datos debe leerse cada uno de los componentes de la estructura:

$leer\_A : \mathbb{I}S \longrightarrow A$
$(is) \longmapsto (leer\_A_1(is), leer\_A_2(is), leer\_A_3(is), ..., leer\_A_n(is))$

En C++:

```
A leer_A(istream& is){
   return crear_A(leer_A1(is), leer_A2(is),
    leer_A3(is), ..., leer_An(is));
};
```

Para el TDA que define los números complejos se tiene:

$leer\_complejo : \mathbb{I}S \longrightarrow \mathbb{C}$
$(is) \longmapsto (leer\_real(is), leer\_real(is))$

En C++ se tiene:

```
double leer_real(istream& is){
   double i;
   is >> i;
   return i;
};

complejo leer_complejo(istream& is){
   return crear_complejo(leer_real(is), leer_real(is));
};
```

Para escribir un TDA, debe escribirse cada uno de los componentes de la estructura en el flujo de datos y retornar el flujo de datos. Se puede definir esta función matemáticamente aprovechando la composición de funciones:

$escribir\_A : \mathbb{O}S \times A \longrightarrow \mathbb{O}S$
$(os, a) \longmapsto escribir\_A_n(escribir\_A_{n-1}(...(escribir\_A_1(os, a_1), ...)a_{n-1}), a_n)$

En C++:

```
ostream& escribir_A(ostream& os, A a){
   return escribir_An(...(escribir_A2(escribir_A_1(os, a_1),
   a2),...), a_n);
};
```

Para la escritura de un número complejo en un flujo se tiene:

$escribir\_complejo : \mathbb{O}S \times \mathbb{C} \longrightarrow \mathbb{O}S$
$(os, z) \longmapsto (escribir\_real(escribir\_real(os, x), y)$

En C++ (se agrega el codigo de escribir real por comodidad):

```
ostream& escribir_real(ostream& os, double x){
   os << x << "\t";
};

ostream& escribir_complejo(ostream& os, complejo z){
   return escribir_real(escribir_real(os, z->x), z->y);
};
```

Debe tenerse en cuenta que para mejorar la escritura del tipo de dato la función escribir real que se definió en la parte de flujos de datos escribe un caracter tabulador al final de cada dato. La declaración de esta función equivale a:

```
ostream& escribir_complejo(ostream& os, complejo z){
   return os << x << "\t" << y << "\t";
};
```

13.5. Funciones Liberadoras

Las funciones liberadoras devuelven la memoria empleada por la estructura del TDA al sistema operativo. En este caso si un tipo de dato A_i es primitivo no requiere liberación si no es primitivo deberá liberarse antes de liberarse la estructura.

En general se tiene:

$liberar\_A : A \longrightarrow \varnothing$
$(a) \longmapsto$

En C++ se tiene:

```
void liberar_A(A a){
   liberar_A1(a->a1);
   liberar_A2(a->a2);
   liberar_A3(a->a3);
   .
   .
   liberar_An(a->an);
   delete a;
};
```

Para el caso de los números complejos, se tiene la siguiente función liberadora:
$liberar\_complejo : \mathbb{C} \longrightarrow \varnothing$
$(z) \longmapsto$

Como el tipo real (`double`) es primitivo, lo unico que se debe liberar es la estructura. En C++ se tiene:

```
void liberar_complejo(complejo z){
delete z;
};
```

13.6. Programa Principal

Tomando la definición de TDA de complejo un sencillo programa que lee dos complejos de la entrada estandar de consola `cin` e imprime en la salida estándar `cout` es:

```
int main(){
    complejo z1 = leer_complejo(cin);
    complejo z2 = leer_complejo(cin);
    complejo z3 = sumar_complejo(z1, z2);
    escribir_complejo(cout, z3);
    liberar_complejo(z1);
    liberar_complejo(z2);
    liberar_complejo(z3);
    system("pause");
    return 0;
};
```

Dado el siguiente flujo de entrada de consola dado por un usuario correspondiente a la suma de dos complejos $(-3, 4i)$ y $(2, -5i)$, el programa retorna a la salida de consola la suma de dos complejos $(-1, -1i)$ como se aprecia a continuación:

```
-3   4
2   -5
-1   -1
Press any key to continue . . .
```

Debe observarse que toda estructura creada después de utilizarse es liberada en el programa principal.

Ejercicios

Bibliografía

Alpuente, M. & Iranzo, P. J. [2007]. *Programación lógica: teoría y práctica*, Pearson Educación.

Deitel, H. M. & Deitel, P. J. [2004]. *Cómo programar en C, C++ y Java*, Pearson Educación.

Iranzo, P. J. [2005]. *Lógica simbólica para informáticos*, Alfaomega.

Johnsonbaugh, R. [2005]. *Matemáticas Discretas*, 6a edn, Pearson.

Kleene, S. [1952]. *Introduction to Metamathematics*, North-Holland, Amsterdam.

Kolman, B., Busby, R. C. & Ross, S. [1997]. *Estructuras de matemáticas discretas para la computación*, Pearson educación, Prentice-Hall Hispanoamericana.

Louden, K. C. [2004]. *Lenguajes de programación, principios y practica*, segunda edn, Thomson.

Rosen, K. H. [2004]. *Matemática discreta y sus aplicaciones*, McGraw-Hill.

Savitch, W. A. [2000]. *Resolucion de problemas Con C++*, 2nd edn, Pearson Educación, México.