Trabajos de Algoritmos y Estructura de Datos

Esp. Ing. José María Sola, profesor.

Revisión 7.1.0
2025-06-02

Tabla de contenidos

Lista de figuras

Lista de tablas

Lista de ejemplos

1

Introducción

El objetivo de los trabajos es afianzar los conocimientos y evaluar su comprensión.

En la sección "Trabajos" de la página del curso[1] se indican cuales de los trabajos acá definidos que son **obligatorios** y cuales **opcionales**, como así también si se deben resolver **individualmente** o en **equipo**.

En el sección "Calendario" de la página del curso[2] se establece cuando es la **fecha y hora límite de entrega**,

Hay trabajos opcionales que son introducción a otros trabajos más complejos, también pueden enviar la resolución para que sea evaluada.

Cada trabajo tiene un **número** y un **nombre**, y su enunciado tiene las siguientes secciones:

1. **Objetivos**: Descripción general de los objetivos y requisitos del trabajo.
2. **Temas**: Temas que aborda el trabajo.
3. **Problema**: *Descripción* del problema a resolver, la *definición completa y sin ambigüedades* es parte del trabajo.
4. **Tareas**: Plan de tareas a realizar.
5. **Restricciones**: Restricciones que deben cumplirse.
6. **Productos**: Productos que se deben entregar para la resolución del trabajo.

[1] https://josemariasola.wordpress.com/aed/assignments/
[2] https://josemariasola.wordpress.com/aed/calendar/

2

Requisitos Generales para las Entregas de las Resoluciones

Cada trabajo tiene sus requisitos particulares de entrega de resoluciones, esta sección indica los requisitos generales, mientras que, cada trabajo define sus requisitos particulares.

Una resolución se considera **entregada** cuando cumple con los **requisitos de tiempo y forma** generales, acá descriptos, sumados a los particulares definidos en el enunciado de cada trabajo.

La entrega de cada resolución debe realizarse a través de *GitHub*, por eso, cada estudiante tiene poseer una cuenta en esta plataforma.

2.1. Requisitos de Forma

2.1.1. Repositorios

En el curso usamos repositorios *GitHub*. Uno público y personal y otro privado para del equipo.

Repositorios público y privado.

```
Usuario
`-- Repositorio público personal para la asignatura
Repositorio privado del equipo
```

Repositorio Personal para Trabajos Individuales

Cada estudiante debe crear un repositorio público dónde publicar las resoluciones de los trabajos individuales. El nombre del repositorio debe ser el de la asignatura. En la raíz del mismo debe publicarse un archivo `readme.md` que actúe como *front page* de la persona. El mismo debe estar escrito en notación *Markdown* y debe contener, como mínimo, la siguiente información:

- Algoritmos y Estructuras de Datos
- Curso.
- Año de cursada, y cuatrimestre si corresponde.
- Legajo.
- Apellido.
- Nombre.

Repositorio personal para la asignatura.

```
Usuario
`-- Repositorio público personal para la asignatura
    `-- readme.md // Front page del usuario
```

Repositorio de Equipo para Trabajos Grupales

A cada equipo se le asigna un **repositorio privado**. En la raíz del mismo debe publicarse un archivo `readme.md` que actúe como *front page* del equipo. El mismo debe estar escrito en notación *Markdown* y debe contener, como mínimo, la siguiente información:

- Algoritmos y Estructuras de Datos
- Curso.
- Año de cursada, y cuatrimestre si corresponde.
- Número de equipo.
- Nombre del equipo (opcional).
- Integrantes del equipo actualizados, ya que, durante el transcurso de la cursada el equipo puede cambiar:

- Usuario *GitHub*.
- Legajo.
- Apellido.
- Nombre.

Repositorio privado del equipo.

```
Repositorio privado del equipo
`-- readme.md // Front page del equipo.
```

2.1.2. Carpetas para cada Resolución

La resolución de cada trabajo debe tener su propia carpeta, ya sea en el repositorio personal, si es un trabajo individual, o en el del equipo, si es un trabajo grupal. El nombre de la carpeta debe seguir el siguiente formato:

> DosDígitosNúmeroTrabajo-NombreTrabajo

O en notación *regex*:

```
[0-9]{2}"-"[a-zA-Z]+
```

Ejemplo 2.1. Nombre de carpeta

00-Hello

En los enunciados de cada trabajo, el número de trabajo para utilizar en el nombre de la carpeta está generalizado con "DD", se debe reemplazar por los dos dígitos del trabajo establecidos en el curso.

Adicionalmente a los productos solicitados para la resolución de cada trabajo, la carpeta debe incluir su propio archivo `readme.md` que actúe como *front page* de la resolución El mismo debe estar escrito en notación *Markdown* y debe contener, como mínimo, la siguiente información:

- Número de equipo.
- Nombre del equipo (opcional).
- Autores de la resolución:
  - Usuario github.
  - Legajo.
  - Apellido.
  - Nombre.
- Número y título del trabajo.
- Transcripción del enunciado.
- Hipótesis de trabajo que surgen luego de leer el enunciado.

Opcionalmente, para facilitar el desarrollo se **recomienda incluir**:

- un archivo `.gitignore`.
- un archivo `Makefile`.footnot:requiered-optional[Para algunos trabajos, el archivo `Makefile` y los tests son obligatorios, de ser así, se indica en el enunciado del trabajo.]
- archivos tests.footnot:requiered-optional[]

Carpeta de resolución de trabajo.

```
Carpeta de resolución de trabajo
|-- .gitignore
|-- Makefile
|-- readme.md // Front page de la resolución
`-- Archivos de resolución
```

Por último, la carpeta **no debe incluir**:

- archivos ejecutables.
- archivos intermedios producto del proceso de compilación o similar.

Ejemplo de Estructura de Repositorios

Ejemplo completo.

```
usuario // Usuario GithHub
`-- Asignatura // Repositorio personal público para a la asignatura
    |-- readme.md // Front page del usuario
    |-- 00-Hello // Carpeta de resolución de trabajo
    |    |-- .gitignore
    |    |-- readme.md // Front page de la resolución
    |    |-- Makefile
    |    |-- hello.cpp
    |    `-- output.txt
    `-- 01-Otro-trabajo
2019-051-02 // Repositorio privado del equipo
|-- redme.md // Front page del equipo
|-- 04-Stack // Carpeta de resolución de trabajo
|    |-- .gitignore
|    |-- readme.md // Front page de la resolución
|    |-- Makefile
|    |-- StackTest.cpp
|    |-- Stack.h
|    |-- Stack.cpp
|    `-- StackApp.cpp
`-- 01-Otro-trabajo
```

2.1.3. Lenguaje de Programación

En el curso se establece la versión del estándar del lenguaje de programación que debe utilizarse en la resolución.

2.1.4. Header Comments (Comentarios Encabezado)

Todo archivo fuente debe comenzar con un comentario que indique el "Qué", "Quiénes", "Cuándo" :

```
/* Qué: Nombre
 * Breve descripción
 * Quiénes: Autores
 * Cuando: Fecha de última modificación
 */
```

Ejemplo 2.2. Header comments

```
/* Stack.h
 * Interface for a stack of ints
 * JMS
 * 20150920
 */
```

2.2. Requisitos de Tiempo

Cada trabajo tiene una **fecha y hora límite de entrega**, los *commits* realizados luego de ese instante no son tomados en cuenta para la evaluación de la resolución del trabajo.

En el calendario del curso[1] se publican cuando es la fecha y hora límite de entrega de cada trabajo.

[1] https://josemariasola.wordpress.com/aed/calendar/

Parte I. Trabajos Introductorios

3

Problemas y Soluciones

Todos los archivos `readme.md` que actúan como *Front Page* de la resolución, deben contener el *Análisis del problema* y el *Diseño de la solución*.

- **Etapa #1: Análisis del Problema**.
    - Transcripción del problema.
    - Refinamiento del problema e hipótesis de trabajo.
    - *Modelo IPO* con:
        - Entradas: nombres y tipos de datos.
        - Proceso: nombre descriptivo.
        - Salidas: nombres y tipos de datos.
- **Etapa #2: Diseño de la solución**. Consta del algoritmo que define el método por el cual el proceso obtiene las salidas a partir de las entradas:
    - Léxico del Algoritmo.
    - Representación visual **ó** textual del Algoritmo.

La resolución incluye archivos fuente que forman el programa que implementan el algoritmo definido. Es importante el programa debe seguir la definición del algoritmo, y no al revés.

4

"Hello, World!" en C++

4.1. Objetivos

- Demostrar capacidad para editar, compilar, y ejecutar programas C mediante el desarrollo de un programa simple. C++.
- Tener un primer contacto con las herramientas necesarias para abordar la resolución de los trabajos posteriores.
- Creación de repositorio personal `git`.
- Armado de equipo de trabajo.

4.2. Temas

- Sistema de control de versiones.
- Lenguaje de programación C++.
- Proceso de compilación.
- Pruebas.

4.3. Problema

Adquirir y preparar los recursos necesarias para resolver los trabajos del curso.

4.4. Restricciones

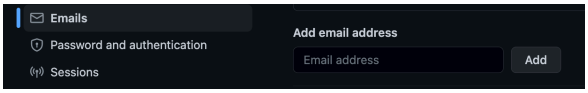
- Ninguna.

4.5. Tarea

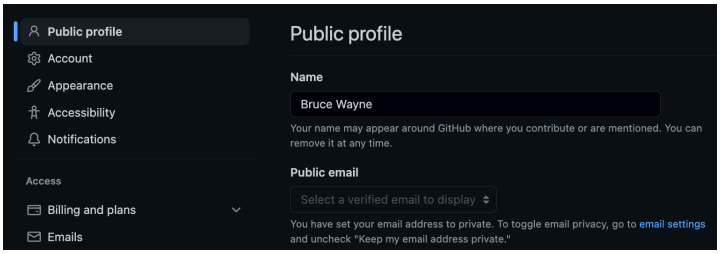
1. Cuenta en *GitHub*

    a. Si no tiene, cree una cuenta *GitHub*.

    b. Si no lo hizo, asocie a su cuenta *GitHub* el email @frba y verifíquelo. Es posible asociar más de una cuenta email a una cuenta *GitHub*.



    c. Si no lo hizo, indique que su cuenta email @frba es **pública**. Esto permite a la cátedra encontrar a los estudiantes. Si por temas de privacidad prefiere no tener como pública esa dirección, puede cambiarla al final del proceso.



2. Repositorio para público para la materia

    a. Cree un repositorio público llamado `AED`.

    b. En la raíz de ese repositorio, escriba el archivo `readme.md` que actúa como *front page del repositorio* personal.

    c. Cree la carpeta `00-CppHelloWorld`.

    d. En esa carpeta, escriba un segundo archivo `readme.md` que actúa como *front page de la resolución*.

3. Compilador

    a. Seleccione, instale, y configure, y pruebe un compilador **C++20** (ó **C++17** ó **C++14** ó **C++11**). Los más osados pueden buscar un compilador que soporte **C++23**.

    b. Registre los resultados anteriores de la siguiente manera:

i. Indique en el `readme.md` el compilador seleccionado, su versión, y la versión de **C++** que compila.

> Es importante separar dos conceptos: la **versión del compilador** de la **versión del lenguaje de programación**. Una versión del compilador compila una o más versiones del lenguaje de programación.
>
> Una forma de conocer la versión del compilador es solicitándolo por línea de comando. Por ejemplo: `g++ --version` ó `clang++ --version`.
>
> Para conocer las versiones del lenguaje de programación que esa versión del compilador compila, se puede consultar la documentación de esa versión del compilador ó experimentar con la opción `-std`.

i. Pruebe el compilador con un programa `hello.cpp` que envíe a `cout` la línea `Hello, World!` o similar.
ii. Ejecute el programa y verifique que la salida es la esperada.
iii. Ejecute el programa con la salida *redireccionada* a un archivo `output.txt`; verifique su contenido.

1. Publicación

   a. Publique el trabajo en el repositorio personal `AED` la carpeta `00-CppHelloWorld` con `readme.md`, `hello.cpp`, y `output.txt`.

2. Armado de Equipo.

   Aunque el trabajo es individual, fomentamos la colaboración entre compañeros para su resolución. Consideramos que es una buena oportunidad para armar equipo para los trabajos siguientes que en su mayoría son grupales. El docente del curso indica la cantidad de integrantes mínima y máxima por equipo.

a. Informe el número de equipo en esta lista[1].

   Con el número de equipo y cuenta @frba, la Cátedra le envía la invitación al repositorio privado del equipo, por eso es importante que su cuenta *GitHub* tenga asociado como email público su email @frba, tal como indica el primer paso.

b. Luego de aceptar la invitación al repositorio privado del equipo, si lo desea, puede cambiar el email público en *GitHub*.

4.6. Productos

```
Usuario                  // Usuario GitHub
`-- AED                  // Repositorio público para la materia
    |-- readme.md        // Archivo front page del usuario
    `-- 00-CppHelloWorld // Carpeta el trabajo
        |-- readme.md    // Archivo front page del trabajo
        |-- hello.cpp    // Archivo fuente del programa
        `-- output.txt   // Archivo con la salida del programa
```

4.7. Referencia

- [CompiladoresInstalacion]
- [BJARNE2013] § 2.2.1 Hello, World!
- [CharacterInputOutputRedirection]
- [Git101]

[1] https://docs.google.com/spreadsheets/d/1v2FFEWdAiUQSeO1i08mHh7O8kyYsgS7_acz95uaS3YQ

5

Resolución de Problemas — Adición

5.1. Objetivos

- Demostrar, mediante un problema simple, el conocimiento de las etapas de resolución de problemas.

5.2. Temas

- Resolución de problemas.
- Entrada de datos.
- Tipos numéricos.
- Adición.
- Léxico.
- Representación de algoritmos.

5.3. Problema

Obtener del usuario dos números y mostrarle la suma.

5.4. Restricciones

- Ninguna.

5.5. Tareas

1. Escribir el archivo `readme.md` que actúa como *front page* de la resolución que contenga lo solicitado en la sección Sección 2.1.2, “Carpetas para cada

Resolución”, y en particular, el *Análisis del Problema* y el *Diseño de la Solución*:

- Etapa #1: Análisis del problema:
  - Transcripción del problema.
  - Refinamiento del problema e Hipótesis de trabajo.
  - Modelo IPO.
- Etapa #2 Diseño de la Solución:
  - Léxico del Algoritmo.
  - Representación del Algoritmo [1]:
    - Representación visual.
    - Representación textual.

2. Escribir, compilar, ejecutar, y probar `Adición.cpp`.

5.6. Productos

```
DD-Adición
 |-- readme.md
 `-- Adición.cpp
```

[1] En este trabajo en particular es necesario presentar ambas representaciones, en el resto de los trabajos se puede optar por una u otra.

6

Valores y Operaciones de Tipos de Datos — Ejemplos

6.1. Objetivos

- Demostrar la aplicación de tipos de datos mediante un programa ejemplo con pruebas.

6.2. Temas

- Valores.
- Operaciones.
- Tipos de datos.
- Representación literal de valores en programas C++.
- `assert`.

6.3. Problema

Diseñar un programa C++ que ejemplifique con pruebas la aplicación de los tipos de datos vistos en clases.

6.4. Restricciones

- Utilice la notación C++ para representar
- No utilice *variables*.
- No extraer valores de `cin`, usar valores literales (constantes).

- No enviar valores a `cout`.

6.5. Tareas

- Este es un *trabajo no estructurado*, que consiste en escribir un programa que ejemplifique el uso de los tipos de datos básicos de C++ vistos en clase: `bool`, `char`, `unsigned`, `int`, `double`, y `string`.

Crédito Extra

> ¿Son los enumerados en la sección anterior realmente todos los tipos que usamos en clase? Justifique.

Ejemplo 6.1. Crédito Extra

Para cada tipo de dato, agregue una notación literal alternativa, si la tiene.

Ejemplo 6.2. Crédito Extra

Intente probar que la suma de diez veces un décimo (`0.1`) es uno (`1.0`). ¿Qué está ocurriendo?

6.6. Productos

```
DD-EjemploTipos
 |-- readme.md
 `-- EjemploTipos.cpp
```

Parte II. Trabajos sobre Funciones

7

Funciones y Comparación de Valores en Punto Flotante — Celsius y Fahrenheit

7.1. Objetivos

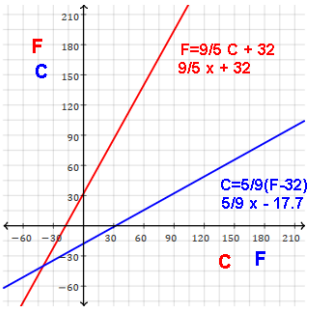
- Demostrar el manejo de funciones y valores punto flotante.

7.2. Temas

- Funciones.
- Tipo `double`.
- División entera y flotante.
- Pruebas con `assert`.
- Argumentos con valor por defecto.
- Introducción a iteraciones con `for`.

7.3. Problema

Se necesita una tabla que presente las temperaturas Celsius convertidas en Fahrenheit, y otra en el sentido opuesto. Use la siguiente imagen, tomada de este artículo[1], como referencia gráfica de las relaciones:



7.3.1. Desafíos con respecto al formato de la tabla

- La columna izquierda muestra temperaturas con valores enteros (i.e, sin décimas de grados).
- La derecha debe tener precisión de una décima.
- Las dos columnas deben estar alineadas a derecha.

7.4. Restricciones

- Se deben construir dos funciones, una para cada conversión que se debe realizar, ambas son $\mathbb{R} \longrightarrow \mathbb{R}$
- Las pruebas deben realizarse con `assert`.

7.5. Análisis

Hay dos sub-problemas que se requieren solucionar antes de poder implementar y probar las funciones.

1. Para implementar la función *Celisus* el problema del valor de la fracción $\frac{5}{9}$ versus la división entera de la expresión `5/9` en C++; y para la *Fahrenheit*, $\frac{9}{5}$ versus `9/5`.

[1] https://www.davidwills.us/math103/linearF/linearFexamples.html

2. Para poder probar las implementación, el problema de la presentación no precisa de los tipos flotantes.

Una solución al primer problema es realizar división entre flotantes.

Para el segundo problema, debemos incorporar la comparación con *tolerancia*, para eso debemos diseñar una función `bool` que reciba dos flotantes a comparar y un flotante que represente la tolerancia y retorna verdedro si los flotantes están cerca según la tolerancia. Se propone la función *AreNear*, y también las soluciones presentadas en [DAWSON2012] y [ERICSON2008].

7.6. Tareas

1. Función *AreNear*:
    a. Escribir el léxico, es decir, la definición matemática de la función.
    b. Escribir las pruebas en `main`.
    c. Escribir el prototipo antes de `main`.
    d. Escribir la definición después de `main`.
    e. Compilar, ejecutar, y evaluar la salida resultante.
2. Función *Celisus* que calcula Celsius a partir de Fahrenheit: mismos pasos.
3. Función *Fahrenheit* que calcula Fahrenheit a partir de Celsius: mismos pasos.
4. (*Opcional*) Escribir en `main` las iteraciones que imprimen las tablas, también pueden ser dos funciones `void`.
5. (*Opcional*) Deasarrolar funciones que grafiquen ambas funciones lineales.

7.7. Productos

```
DD-Celsius
 |-- readme.md
 `-- Temp.cpp
```

8

Funciones y Operador Condicional

8.1. Objetivos

- Demostrar manejo de funciones y del operador condicional.

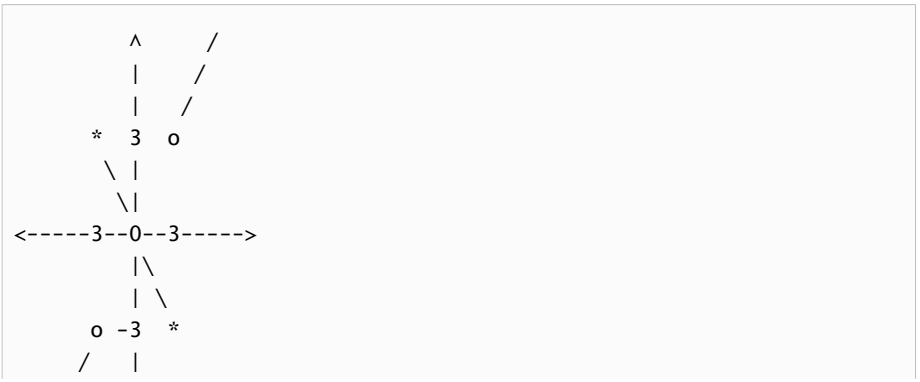
8.2. Temas

- Operador condicional.
- Funciones.

8.3. Problema

Desarrollar las siguientes funciones:

1. Valor absoluto.
2. Valor mínimo entre dos valores.
3. Función $f3$, definida por:



```
   /    |
  /     v
```

8.4. Restricciones

- Las pruebas deben realizarse con `assert`.
- Cada función debe aplicar el operador condicional.

8.5. Tareas

Por cada función:

1. Escribir el léxico, es decir, la definición matemática de la función.
2. Escribir las pruebas.
3. Escribir los prototipos.
4. Escribir las definiciones.

8.6. Productos

```
DD-Cond
 |-- readme.md
 |-- Abs.cpp
 |-- Min.cpp
 `-- F3.cpp
```

9

Precedencia de Operadores — Bisiesto

9.1. Objetivos

- Demostrar el uso de operadores booleanos y expresiones complejas.

9.2. Temas

- Expresiones.
- Operadores booleanos: `and`, `or`, y `not`.
- Operador resto: `%`.
- Asociatividad de Operadores: ID ó DI.
- Precedencia de Operadores.
- Orden de evaluación de Operandos.
- Funciones.
- Precondiciones.

9.3. Problema

Desarrollar una función que dado un año, determine si es bisiesto.

9.4. Restricciones

- El nombre de la función debe ser `IsBisiesto` [1].
- Aplicar operadores booleanos
- No aplicar expresión condicional (`?:`, es decir, el *operador ternario*).
- No aplicar sentencia condicional (`if`, `if-else`, ni `switch`).
- Las pruebas deben realizarse con `assert`.

9.5. Tareas

1. Escribir en `readme.md` la definición matemática de la función, y, si tiene, incluir las precondiciones.
2. Escribir en `IsBisiesto.cpp` las pruebas.
3. Escribir en `IsBisiesto.cpp` el prototipo.
4. Escribir en `IsBisiesto.cpp` la definición.
5. Incluir en `readme.md` el *árbol de expresión* asociado a la expresión de retorno de la función.

9.6. Créditos Extra

9.6.1. Conceptos de Expresiones

Explique en `readme.md` los siguientes conceptos, haciendo referencia al árbol de expresión:

1. Asociatividad de Operadores.
2. Precedencia de Operadores.
3. Orden de evaluación de Operandos.

9.6.2. Sentencia de Selección

Desarrollar en `IsBisisestoIf.cpp` una nueva versión de `IsBisiesto` que en vez de operadores de conexión lógica (i.e., `and`, `or`) utilice `if` o `if-else`. Analizar

[1] Es una práctica común uilizar el prefijo `Is` para predicados, es decir, funciones que retornan un valor lógico.

comparativamente las dos implementaciones e identificar pros y cons de cada una.

9.6.3. Tipo Año

Desarrollar en `IsBisisestoY.cpp` una nueva versión de `IsBisiesto` que en vez de recibir un natural reciba un *año*. Las ecuaciones se verían de de esta manera:

```
assert( not IsBisiesto(1582y) );
assert(     IsBisiesto(2020y) );
```

Explique el sufijo `y`.

9.6.4. Biblioteca Estándar

Investigue si la biblioteca estándar provee una funcionalidad similar a `IsBisiesto`, si es así, desarrolle en `IsBisisestoStd.cpp` un programa que pruebe esa funcionalidad.

9.7. Productos

```
DD-Bisiesto
 |-- readme.md
 |-- IsBisiesto.cpp
 |-- IsBisiestoIf.cpp  // Opcional
 |-- IsBisiestoY.cpp   // Opcional
 `-- IsBisiestoStd.cpp // Opcional
```

10

Funciones Recursivas con Operador Condicional

10.1. Objetivos

- Demostrar manejo de funciones definidas recursivamente e implementadas con el operador condicional.

10.2. Temas

- Funciones recursivas.
- Operador condicional.

10.3. Problema

Desarrollar las siguientes funciones:

1. División entera de naturales: `Div`.
2. MCD (Máximo Común Denominador): `Mcd` [PINEIRO].
3. Factorial: `Fact`.

Un número factorial puede ser muy grande, por eso hay que elegir el tipo de la función correctamente.

4. Fibonacci: `Fib`.

Notar que esta función es doblemente recursiva.

10.4. Restricciones

- Las pruebas deben realizarse con `assert`.
- Cada función debe aplicar el operador condicional.

10.5. Tareas

Por cada función:

1. Escribir el léxico, es decir, la definición matemática de la función.
2. Escribir las pruebas.
3. Escribir los prototipos.
4. Escribir las definiciones.

10.6. Productos

```
DD-Recur
 |-- readme.md
 |-- Div.cpp
 |-- Mcd.cpp
 |-- Factorial.cpp
 `-- Fibonacci.cpp
```

11

Sistema de Funciones — Días del Mes

11.1. Objetivos

- Demostrar el uso de funciones para resolver problemas.

11.2. Temas

- Expresiones.
- Expresión condicional.
- Especificación de funciones.
- Funciones puras.
- Precondiciones.
- Poscondiciones.
- Funciones que invocan funciones.
- Efecto de lado de una expresión.
- Transparencia referencial.
- Entrada estándar.
- Pruebas y aplicación.

11.3. Problema

Desarrollar un programa que informe la cantidad de días de un mes.

11.4. Restricciones

- Los datos se extraen de la entrada estándar.
- El año se pide solo para Febrero.
- La solución debe basarse en la función pura `GetCantidadDeDías`.
- La función `GetCantidadDeDías` debe invocar `IsBisiesto` (ver trabajo Precedencia de Operadores — Bisiesto).
- La función `main` tiene dos responsabilidades: probar `GetCantidadDeDías` y resolver el problema con una aplicación.
- La prueba y la aplicación se hacen desde funciones invocadas por `main` o en bloques distintos delimitado por llaves `{` y `}`.
- Las pruebas deben realizarse con `assert`.
- Desarrollar funciones no puras para la extracción de datos de la entrada estándar y para el envío de resultados a la salida estándar.
- Aplicar expresiones condicionales y sentencias condicionales según corresponda.

11.5. Tareas

1. Escribir la especificación matemática de la función `GetCantidadDeDías`.
2. Escribir las pruebas.
3. Escribir el prototipo.
4. Escribir la definición.
5. Diseñar las funciones de la aplicación.
6. Escribir la aplicación.

11.6. Productos

```
DD-CantidadDeDiasDelMes
 |-- readme.md
 `-- CantidadDeDiasDelMes.cpp
```

Parte III. Trabajos sobre Control de Flujo de Ejecución

Expresiones y Valores versus Sentencias y Acciones

12

Mayor de dos Números

12.1. Problema

Dado dos números informar cuál es el mayor.

12.2. Productos

- Sufijo del nombre de la carpeta: `Mayor`
- `readme.md`.
- `Mayor.cpp`.

13

Repetición de Frase

13.1. Problema

Enviar una frase a la salida estándar muchas veces.

13.2. Restricciones

Realizar dos versiones del algoritmo y una implementación para cada uno:

- Salto condicional.
- Iterativa esctructurada.
- Retorna un string largo
- do while, while, for, goto

13.3. Productos

- Sufijo del nombre de la carpeta: `Repetición`
- `readme.md` con los dos algoritmos.
- `Saltos.cpp`.
- `Iteración.cpp`.

Parte IV. Trabajos sobre Abstracción con Tipos

14

Enumeraciones

1. Escriba un programa que declare una variable que pueda almacenar cualquier punto cardinal.
2. Extender el programa de la sección programa de la sección 1.5. Funciones que Retornan o Reciben Tipos Enum del texto "Enumeraciones"[1] para que contenga una función que dado un día y turno, informe la asignatura que debemos cursar.

14.1. Productos

```
DD-Enum
 |-- Cardinal.cpp
 `-- SemanaDeCursada.cpp
```

[1] https://josemariasola.wordpress.com/aed/papers/#Enums

15

Uniones

1. Escriba un programa ejemplo que opere sobre dos variables:
   - una que almacene tanto enteros (`int`s) como naturales (`unsigned`s).
   - y otra que almacene tanto caracteres (`char`s) como reales (`double`s).
2. Extender el programa Caninos del texto "Uniones"[1] para que incluya las siguientes variables:
   a. `Santas`[2]
   b. `WileE`[3]
      i. ¿El cambio es simplemente agregar una variable?
   c. `Snowball2`[4] y `Simba`[5]
      i. ¿El cambio es simplemente agregar dos variables?
      ii. ¿Deberían existir en el programa conceptos como `Mamífero` ó `Carnívoro`?

15.1. Productos

```
DD-Union
 |-- EjemploDeUniones.cpp
```

[1] https://josemariasola.wordpress.com/aed/papers/#Unions
[2] https://en.wikipedia.org/wiki/Santa%27s_Little_Helper
[3] https://en.wikipedia.org/wiki/Wile_E._Coyote_and_the_Road_Runner
[4] https://en.wikipedia.org/wiki/Simpson_family#Snowball_II
[5] https://en.wikipedia.org/wiki/Simba

```
`-- Animados.cpp
```

16

Problemas, Arrays, String & Enumeraciones — CUIL

16.1. Objetivos

- Demostrar capacidad de definición de problemas y de diseño de implementación, con clara separación entre el *dominio del problema* y el *dominio de la solución*.
- Aplicar secuencias y ciclos de iteración.
- Definir nuevos tipos de datos enumerados.

16.2. Temas

- Dominio del problema.
- Modelo IPO.
- Dominio de la solución.
- Definición de conjunto de valores con `enum struct`. [Enums]
- Strings como secuencia de valores.
- Secuencia de valores y `array`.
- Ciclo de iteración `for` y sus variantes.
- Operación `•.at(•)`.

16.3. Problema

El problema en sí es definir, acotar y refinar el problema a resolver; para luego implementar la solución. Partimos de esta frase:

Se necesita crear CUILS de personas físicas.

16.4. Restricciones

- La definición del problema debe estar en `readme.md` y se debe aplicar *Modelo IPO*.
- Para la implementación debe aplicarse:

    1. Funciones.
    2. `enum`.
    3. `string`.
    4. peración `•.at(•)` y no `•[•]`.
    5. `assert`.

16.5. Tareas

1. Especificar el problema en `readme.md`, incluir un modelo IPO y las restricciones que el equipo decida.
2. Diseñar un set de pruebas.
3. Especificar la solución en `readme.md`, incluir:

    - descripción y reestricciones generales del producto solución,
    - los *principales* funciones con sus precondiciones y sus poscondiciones,
    - y tipos de datos utilizados.
4. (*Opcional*) Especificar matemáticamente la función más importante en `readme.md`.
5. Diseñar y codificar las pruebas en `main`.
6. Declarar los prototipos de antes de `main`.
7. Implementar las funciones.

16.6. Productos

```
DD-Cuil
 |-- readme.md
 `-- Cuil.cpp
```

17

Arreglos & Dimensiones — Total de Ventas

17.1. Objetivos

- Demostrar capacidad de construcción de tipos compuestos mediante la aplicación sucesiva de producto cartesiano.
- Aplicación de ciclo *for clásicos* y de ciclo *for de intervalos* (*range_for*).
- Aplicación de redireccionamiento de los flujos estándar.

17.2. Temas

- Producto cartesiano.
- Secuencias finitas.
- Tipos `std::array<T,N>`.
- Inferencia de tipo con `auto`.
- Streams (*flujos*).
- Redirección de entrada y salida.
- Interfaz fluida.
- for clásico: `for (` *sentencia-inic* *condición*$_{opc}$ `;` *expresión*$_{opc}$ `)` *sentencia*
- for intervalo: `for (` *sentencia-inic*$_{opc}$ *declaración-for-intervalo* `:` *inicializador-for-intervalo* `)` *sentencia*

17.3. Problema

Esta es una serie de problemas que parten de una necesidad general: *“Dado los importes, mostrar las ventas totales”*, y que después se particulariza en necesidades puntuales:

- Necesidad #1: Dado los importes, mostrar ventas totales.
- Necesidad #2: Dado los importes y meses (de 0 a 11), mostrar ventas totales por mes.
- Necesidad #3: Dado los importes, meses, y números de los tres vendedores (0, 1, 2), mostrar total de ventas por mes y vendedor.
- Necesidad #4: Dado los importes, meses, números de los tres vendedores, y números de las cuatro regiones (0, 1, 2, 4), mostrar total de ventas por mes, vendedor, y región.

17.4. Restricciones

- Se deben crear archivos con set de datos de diferentes para cada solución:
  - `Test0.txt`
  - `Test1.txt`
  - `Test2.txt`
  - `Test3.txt`
- Los datos se extraen de `cin`, no vienen en ningún orden en particular, los importes son enteros, el resultado se envía a `cout`
- Aplicar `std::array<T,N>` y no `T[N]`.
- Aplicar operación `•.at(•)` y no `•[•]`.
- Total de importes: Las cuatro necesidades rondan en presentar el total de importes, hay varias soluciones posibles, las que se deben implementar son las que usan una única variable de múltiples dimensiones, son las que están marcadas con una estrella (★) a continuación.
  - Soluciones posibles a la necesidad #1:
    - ★ 1 variable entera (cero dimensiones)

- Soluciones posibles a la necesidad #2:
  - 12 variables enteras (cero dimensiones, no aprovecha patrón)
  - ★ 1 variable arreglo de 12 enteros (una dimensión, aprovecha patrón)
- Soluciones posibles a la necesidad #3:
  - 6 variables enteras
  - 3 variables arreglos de 12 enteros
  - ★ 1 variable arreglo de 3 arreglos de 12 enteros (dos dimensiones)
- Soluciones posibles a la necesidad #4:
  - 144 variables enteras
  - 12 variables arreglo de 12 enteros
  - 4 variables arreglos de 3 arreglos de 12 enteros
  - ★ 1 variable arreglo de 4 arreglos de 3 arreglos de 12 enteros (tres dimensiones)

- Cada solución debe estar en archivos fuente diferentes:
  - `Dim0.cpp`
  - `Dim1.cpp`
  - `Dim2.cpp`
  - `Dim3.cpp`

Crédito Extra

Los siguientes son ítems opcionales que se basan en la necesidad #4, y deben resolverse en `Dim3Extra.cpp`:

- Implementar las funciones que permitan representar a los vendedores con *strings* y las regiones con *enum* (Norte, Sur, Este, y Oeste), en vez de representarlos con números.
- Implementar las funciones *LeerDatos* y *MostrarTotales*
- Presentar las tablas lo más claro posible con formato, alineación numérica y con títulos.

- Agregar estadísticas, por lo menos una que aplique máximo, otra mínimo, y otra promedio. Por ejemplo: *GetVendedorConMasVentas(mes, región)*.
- Describir en `readme.md` las ventajas y desventajas de aplicar:
  - *for-intervalo* en vez de *for clásico*.
  - `std::array<T,N>` en vez de `T[N]`.
  - `•.at(•)` en vez de `•[•]`.

17.5. Tareas

Por cada necesidad:

1. Diseñar el set de datos para la prueba.
2. Implementar la solución.
3. Ejecutar la solución con redirección de la entrada para que lea del set de datos de prueba. Por ejemplo: `./Dim0 < Test0.txt`

17.6. Productos

```
DD-Dims
 |-- readme.md
 |-- Test0.txt
 |-- Dim0.cpp
 |-- Test1.txt
 |-- Dim1.cpp
 |-- Test2.txt
 |-- Dim2.cpp
 |-- Test3.txt
 |-- Dim3.cpp
 `-- Dim3Extra.cpp
```

18

Diagonal de una Matriz

18.1. Objetivos

- Escribir un programa que determine la suma de la diagonal de una matriz.

18.2. Restricciones

- La suma la debe calcular una función que tenga como parámetro *in* una matriz.

18.3. Productos

```
DD-DiagonalMatriz
 |-- readme.md
 `-- DiagonalMatriz.cpp
```

19

Tipo Color

19.1. Objetivos

- Demostrar capacidad de construcción de tipos compuestos basados en tipos existentes y simples, es decir, no compuestos.

19.2. Temas

- Tipo de dato definido por el usuario (programador).
- Tipo Abstracto de Datos.
- Especificación.
- Implementación.
- Definición de conjunto de valores con `struct`.
- Tipos enteros de ancho fijo.
- Variables externas.
- Variables `const`.

19.3. Problema

Diseñar un tipo Color basado en el modelo RGB[1], con tres canales de 8 bits. Todo color está compuesto por tres componentes: intensidad de *red* (rojo), de *green* (verde), y de *blue* (azul). Cada intensidad está en el rango [0, 255]. Definir los valores para rojo, azul, verde, cyan, magenta, amarillo, negro, y blanco. Dos

[1] https://en.wikipedia.org/wiki/RGB_color_model

colores se pueden mezclar, lo cual produce un nuevo color que tiene el promedio de intensidad para cada componente.

Crédito Extra

La operación *Mezclar* mezcla en partes iguales; desarrollar una variante de la operación que permita indicar las proporciones de las partes.

Crédito Extra

Desarrollar la operaciones *Sumar* y *Restar* que dados dos colores suma o resta la intensidad de cada canal, siempre dando resultados en el rango [0, 255]. Utilizá estas operaciones para inicializar los colores secundarios, blanco, y negro.

Crédito Extra

Desarrollar la operación *GetComplementario* que dado un color obtiene el complementario u opuesto. Por ejemplo, el complementario de rojo es cyan.

Crédito Extra

Desarrollar la operación *GetHtmlHex* que genera un string con la representación hexadecimal para HTML de un color. Por ejemplo, `assert( "#0000ff" == GetHtmlHex(azul) );`

Crédito Extra

Desarrollar la operación *GetHtmlRgb* que genera un string con la representación rgb para HTML de un color. Por ejemplo `assert( "rgb(0,0,255)" == GetHtmlRgb(azul) );`

Crédito Extra

Codificar la función `CrearSvgConTextoEscritoEnAltoContraste` que dado un nombre archivo sin extensión, un texto, y un color de letra

genera un archivo SVG[2] con el texto en un color y fondo en su complementario.

Por ejemplo `CrearSvgConTextoEscritoEnAltoContraste("Mensaje", "¡Hola, Mundo!", cyan)` genera el archivo `Mensaje.svg` con el siguiente contenido:

```
<svg xmlns="http://www.w3.org/2000/svg">
 <rect x="0" y="0" height="30" width="120"
 style="fill: #ff0000"/>
 <text x="5" y="18" style="fill:
 rgb(0,255,255);background-color: #ff0000">
  ¡Hola, Mundo!
 </text>
</svg>
```

Que se visualiza así:

Notar que el fondo tiene el color complementario del texto y que, tan solo por fines ilustrativos, el color de fondo se establece en notación hexadecimal, y el color del texto en notación rgb.

19.4. Restricciones

- Las operaciones de proyección para *red*, *green*, y *blue* se implementan con acceso directo a los componentes, no es necesario definir *getters* especiales. Por la misma razón, los *setters* no son necesarios.
- Utilizar el tipo `uint8_t` de `cstdint`, si no es posible, usar `unsigned char`.
- Los colores primarios, secundarios, negro y blanco deben implementarse como ocho variables declaradas fuera de `main` y de toda función, con el calificador `const` para que no puedan modificarse.

[2] https://en.wikipedia.org/wiki/Scalable_Vector_Graphics

- Implementar la operación `IsIgual` que retorna `true` si un color es igual a otro, si no, `false`.

Crédito Extra

> Responder en `readme.md` porqué se debe usar `uint8_t`. Si tu compilador no te permite usar `uint8_t`, indicar porqué es correcto usar `unsigned char` pero no `char`.

19.5. Tareas

1. Especificar matemáticamente el tipo en `Color.md`:
   a. Especificar el conjunto de operaciones.
   b. Especificar el conjunto de valores.
2. Diseñar y codificar las pruebas en `main`.
3. Declarar los prototipos de las operaciones arriba de `main`.
4. Declarar `Color` antes de los prototipos las operaciones.
5. Compilar: Luego de finalizar tareas anteriores, estamos en condiciones de compilar. Deberíamos obtener error de *linkeo* (i.e., vinculación) pero no de compilación.
6. Codificar las definiciones de las operaciones, debajo de `main`.
7. Probar: Luego de las definiciones, deberíamos poder realizar el proceso de traducción completo (i.e., compilación y linkeo) sin errores. Una vez obtenido el programa ejecutable, deberíamos poder ejecutarlo sin errores.

19.6. Productos

```
DD-Color
 |-- readme.md
 |-- Color.md       // Especificación
 `-- Color.cpp      // Implementación y pruebas
```

Crédito Extra

> Estructurar la solución con separación física en archivos de:

- pruebas,
- de parte pública de la implementación, y
- de parte privada de la implementación.

Escribir un `makefile` que construya y pruebe la solución. Estos temas están desarrollados en [Interfaces-Make]

```
DD-Color
 |-- readme.md
 |-- Makefile
 |-- Color.md       // Especificación
 |-- Color.h        // Implementación: Parte Pública
 |-- ColorTest.cpp  // Pruebas
 `-- Color.cpp      // Implementación: Parte Privada
```

20

Especificación del Tipo de Dato Fecha

20.1. Tarea

Especificar el tipo de dato "Fecha", lo cual implica especificar su conjunto de valores y su conjunto de operaciones sobre esos valores.

20.2. Productos

- `readme.md`:
  - Conjunto de Valores.
  - Conjunto de Operaciones.

Parte V. Trabajos sobre Abstracción con Tipos — Caso de Estudio: Geometría

21

Enumeraciones & Estructuras — Plano y Punto

21.1. Objetivos

- Demostrar capacidad básica de construcción de tipos compuestos basados en tipos existentes y simples, es decir, no compuestos.

21.2. Temas

- Tipo de dato definido por el usuario (programador).
- Definición de conjunto de valores con `struct`.
- Definición de conjunto de valores con `enum struct`.

21.3. Problema

Dado un punto determinar en qué parte en del plano está:

- Cuadrante I, II, II, IV.
- Eje X o Eje Y.
- Origen (en los dos ejes).

21.4. Restricciones

- La solución debe ser una función que reciba un valor del tipo `Punto` y retorne un valor del tipo `ParteDelPlano`. Posibles nombres de la función:
  - `DóndeEstá`

    - `GetParteDelPlano`
    - `DondeEnElPlanoEstá`
- `Punto` debe implementarse con `struct`.
- `ParteDelPlano` debe implementarse con `enum struct`.

Crédito Extra

> Especificar e implementar las operaciones *distancia entre dos puntos* y *distancia al origen*.
>
> Analizar ventajas y desventajas de definir una operación en función de la otra.

21.5. Tareas

1. Debatir Las diferentes formas matemáticas de definir puntos en el plano y las diferentes formas de representarlos en memoria.
2. Especificar matemáticamente el tipo *ParteDelPlano* en `ParteDelPlano.md`:
    a. Especificar el conjunto de operaciones.
    b. Especificar el conjunto de valores.
3. Especificar matemáticamente el tipo *Punto* en `Punto.md`:
    a. Especificar el conjunto de operaciones.
    b. Especificar el conjunto de valores.
4. Diseñar y codificar las pruebas en `main`.
5. Declarar los prototipos de las operaciones antes de `main`.
6. Declarar `Punto` y `ParteDelPlano` antes de los prototipos las operaciones.
7. Compilar: Luego de finalizar tareas anteriores, estamos en condiciones de compilar. Deberíamos obtener error de *linkeo* (i.e., vinculación) pero no de compilación.
8. Codificar las definiciones de las operaciones, luego de `main`.
9. Probar: Luego de las definiciones, deberíamos poder realizar el proceso de traducción completo (i.e., compilación y linkeo) sin errores. Una vez obtenido el programa ejecutable, deberíamos poder ejecutarlo sin errores.

21.6. Productos

```
DD-Plano
 |-- readme.md
 |-- ParteDelPlano.md // Especificación
 |-- Punto.md         // Especificación
 `-- Plano.cpp        // Implementación y pruebas
```

22

Tipo Círculo

22.1. Objetivos

- Demostrar capacidad de construcción de tipos compuestos basados en otros tipos existentes o creados por el programador.

22.2. Temas

- Tipo de dato definido por el usuario (programador).
- Tipo Abstracto de Datos.
- Especificación.
- Implementación.
- Definición de conjunto de valores con `struct`.

22.3. Problema

Diseñar un tipo Círculo en el plano, con las operaciones:

- GetCircunferencia
- GetÁrea
- IsDentro, que dado un punto determina si está dentro de un círculo.
- Mover, que traslada el círculo a otro lugar del plano.

22.3.1. Analizar y Comparar

- Las diferentes formas matemáticas de definir círculos y las diferentes formas de representarlos en memoria.
- ¿Podría representarse con un `array` y con `using`?

22.4. Productos

```
DD-Círculo
 |-- readme.md
 |-- Círculo.md       // Especificación
 `-- Círculo.cpp      // Implementación y pruebas
```

23

Tipo Triángulo

23.1. Objetivos

- Demostrar capacidad de construcción de tipos compuestos basados en otros tipos existentes o creados por el programador.

23.2. Temas

- Tipo de dato definido por el usuario (programador).
- Tipo Abstracto de Datos.
- Especificación.
- Implementación.
- Definición de conjunto de valores con `struct`.

23.3. Problema

Diseñar un tipo Triángulo en el plano, con las operaciones:

- GetPerímetro
- GetÁrea
- IsEscaleno
- IsEquilátero
- IsIsósceles
- GetTipoPorLados
- GetTipoPorÁngulos

- GetCentro

23.3.1. Analizar y Comparar

- Las diferentes formas matemáticas de definir triángulos y las diferentes formas de representarlos en memoria.
- ¿Podría representarse con un `array` y con `using`?

23.4. Restricciones

- Los dos conjuntos de tipos se deben implementar con un `enum struct`.

23.5. Productos

```
DD-Triángulo
 |-- readme.md
 |-- Triángulo.md       // Especificación
 `-- Triángulo.cpp      // Implementación y pruebas
```

24

Estructuras & Arreglos — Cuadriláteros y Rectángulos

24.1. Objetivos

- Demostrar capacidad de construcción de tipos compuestos basados en otros tipos existentes o creados por el programador.
- Demostrar capacidad de selección criterios de la representación de tipos en memoria.

24.2. Temas

- Tipo de dato definido por el usuario (programador).
- Definición de conjunto de valores con `struct`.
- Definición de conjunto de valores con `using`.

24.3. Problemas

1. Determinar el perímetro de un cuadrado.
2. Determinar si un punto está en un cuadrado.
3. Determinar el perímetro de un rectángulo.
4. Determinar si un cuadrado está dentro de un rectángulo.
5. Determinar si un rectángulo es cuadrado.
6. Determinar el perímetro de un cuadrilátero.
7. Determinar si un cuadrilátero está dentro de un rectángulo.

Todos las figuras tienen ubicación en el plano, y los cuadrados y rectángulos tienen sus lados paralelos a los ejes.

24.3.1. Analizar y Comparar

- Las diferentes formas matemáticas de definir rectángulos, cuadrados, y cuadriláteros en el plano y las diferentes formas de representarlos en memoria.

24.4. Restricciones

- Las solución deben implementarse con funciones que reciban un valores del tipo `Punto`, `Cuadrado`, `Rectángulo`, ó `Cuadrilátero`. Posibles nombres de las funciones:
  - `IsDentro`
  - `GetPerímetro`
  - `IsCuadrado`

Crédito Extra

> Especificar e implementar las operación *como cuadrado* (`AsCuadrado`) que recibe un valor del tipo `Rectángulo` y retorna el `Cuadrado` equivalente. ¿Qué precondiciones pondría?

24.5. Tareas

1. Especificar matemáticamente el tipo en `Cuadrado.md`:
   a. Especificar el conjunto de operaciones.
   b. Especificar el conjunto de valores.
2. Especificar matemáticamente el tipo en `Rectángulo.md`:
   a. Especificar el conjunto de operaciones.
   b. Especificar el conjunto de valores.
3. Especificar matemáticamente el tipo en `Cuadrilátero.md`:

   a. Especificar el conjunto de operaciones.
   b. Especificar el conjunto de valores.
4. Diseñar y codificar las pruebas en `main`.
5. Declarar los prototipos de las operaciones antes de `main`.
6. Declarar los tipos antes de los prototipos las operaciones.
7. Compilar: Luego de finalizar tareas anteriores, estamos en condiciones de compilar. Deberíamos obtener error de *linkeo* (i.e., vinculación) pero no de compilación.
8. Codificar las definiciones de las operaciones, luego de `main`.
9. Probar: Luego de las definiciones, deberíamos poder realizar el proceso de traducción completo (i.e., compilación y linkeo) sin errores. Una vez obtenido el programa ejecutable, deberíamos poder ejecutarlo sin errores.

24.6. Productos

```
DD-Cuadrilátero
 |-- readme.md
 |-- Cuadrado.md       // Especificación
 |-- Rectángulo.md     // Especificación
 |-- Cuadrilátero.md   // Especificación
 `-- Cuadriláteros.cpp // Implementación y pruebas
```

Parte VI. Trabajos sobre Estructuras Dinámicas — Caso de Estudio: Geometría II

Oxymoron: *"Estructura Dinámica"*. ¿Existe tal cosa?

25

Simulación de Estructuras Dinámicas — Polígonos

25.1. Objetivos

- Demostrar capacidad de reconocer estructuras dinámicas.
- Demostrar capacidad construcción de estructuras dinámicas.

25.2. Temas

- Tipo de dato definido por el usuario (programador).
- Estructura dinámica.
- Reserva manual explícita de memoria y *heap*.
- Operadores *new* y *delete*.
- Punteros.

25.3. Problema

Determinar el perímetro de un polígono en el plano.

La cantidad de vértices puede diferir para cada polígono.

25.3.1. Análisis y Solución

¿Cuál es la forma más básica y simple de definir matemáticamente un polígono?

Secuencia de vértices, cada vértice es un punto en el plano. El orden de la secuencia determina las aristas o lados del polígono

¿Cómo podemos representar en memoria secuencias de puntos de longitud variable?

Solución #1 — Representación contigua mediante arreglo

Indicar ventajas y desventajas.

- Solución #1a: Indicar fin de secuencia con la repetición del primer o último punto.
- Solución #1b: Indicar fin de secuencia con un natural que indica la cantidad de vértices.

Indicar ventajas y desventajas de 1a y 1b.

Solución #2 — Representación enlazada mediante nodos y punteros a siguientes nodos.

Indicar ventajas y desventajas.

25.4. Operaciones

- AddVértice
- GetVértice
- SetVértice
- RemoveVértice
- GetCantidadLados
- GetPerímetro

25.5. Restricciones

Las pruebas, en particular los `asserts` deben ser iguales para ambas implementaciones.

25.6. Tareas

1. Especificar matemáticamente el tipo en `Polígono.md`:
    a. Especificar el conjunto de operaciones.
    b. Especificar el conjunto de valores.
2. Diseñar y codificar las pruebas en `main`.
3. Declarar los prototipos de las operaciones antes de `main`.
4. Declarar los tipos antes de los prototipos las operaciones.
5. Compilar: Luego de finalizar tareas anteriores, estamos en condiciones de compilar. Deberíamos obtener error de *linkeo* (i.e., vinculación) pero no de compilación.
6. Codificar las definiciones de las operaciones, luego de `main`.
7. Probar: Luego de las definiciones, deberíamos poder realizar el proceso de traducción completo (i.e., compilación y linkeo) sin errores. Una vez obtenido el programa ejecutable, deberíamos poder ejecutarlo sin errores.

25.7. Productos

```
DD-Plano
 |-- readme.md
 |-- Polígono.md      // Especificación
 |-- PolígonoCont.cpp // Implementación y pruebas
 `-- PolígonoLink.cpp // Implementación y pruebas
```

26

Interfaces & Implementaciones — Modularización

Reestructurar las tipos construidos en los anteriores trabajos para que tengan separación física en archivos:

- de especificación,
- de pruebas,
- de parte pública de la implementación,
- de parte privada de la implementación, y
- de construcción (`Makefile`).

Escribir un `Makefile` para cada tipo que lo construya y pruebe, y escribir un `Makefile` que construya todo. Estos temas están desarrollados en [Interfaces-Make]

Crédito Extra

> Lo referido a `Makefile`s es opcional

Los tipos son:

- `Punto`
- `ParteDelPlano`
- `Círculo`
- `Triángulo`

- `Cuadrado`
- `Cuadrilátero`
- `Polígono`

Cada tipo debe tener su propia subcarpeta dentro del trabajo.

```
DD-Modularización
 |-- readme.md
 |-- Makefile           // Construye todo usando los Makefiles de las
 subcarpetas
 |-- Tipo               // Carpeta contenedora del tipo
 |    |-- Makefile      // Construye y prueba tipo
 |    |-- Tipo.md       // Especificación de tipo
 |    |-- Tipo.h        // Implementación: Parte Pública, Interfaz o
 Contrato
 |    |-- TipoTest.cpp  // Pruebas
 |    `-- Tipo.cpp      // Implementación: Parte Privada
 |-- Más tipos...
 :
```

27

Geometría — Desarrollo de Tipos

27.1. Introducción

Este trabajo se hace uso del Capítulo 19, *Tipo Color* y es el primero de una secuencia de trabajos que aplican tipos para solucionar problemas de geometría.

Este trabajo tiene como tema central la construcción de tipos mediante producto cartesiano; el tema se desarrolla en [Structs-Arrays].

27.2. Objetivos

- Demostrar capacidad de construcción de tipos compuestos basados en otros tipos, simples o compuestos, existentes o nuevos.

27.3. Temas

- Tipo de dato definido por el usuario (programador).
- Tipo Abstracto de Datos.
- Especificación.
- Implementación.
- Definición de conjunto de valores con `struct`.
- Definición de conjunto de operaciones con funciones y pasaje de argumentos por referencia (i.e., variable).
- Estructura dinámica con capacidad máxima.

27.4. Problema

Construir el tipo Polígono con color. Un polígono tiene una cantidad dinámica de vértices, y el tipo debe incluir las operaciones para agregar, remover, acceder y modificar esos vértices.

Crédito Extra

> Los tipos y operaciones marcados como opcionales son crédito extra. También podes agregar las operaciones que quieras.

Tipo	Valores	Operaciones
Punto	Representa un punto en el plano con coordenadas cartesianas.	• IsIgual • GetDistancia • GetDistanciaAlOrigen • GetRho (*opcional*) • GetPhi (*opcional*) • GetCuadrante (*opcional*) • GetEje (*opcional*) • GetSemiplano (*opcional*) • Mover (*opcional*)
Círculo (*opcional*)	Representa un círculo con color en el plano.	• GetCircunferencia • GetÁrea • Mover
Triángulo (*opcional*)	Representa triángulos con color en el plano, se lo describe por tres puntos y su color.	• GetPerímetro • GetÁrea • IsEscaleno • IsEquilátero • IsIsósceles • GetTipo (*opcional*) • GetCentro (*opcional*)

Tipo	Valores	Operaciones
Rectángulo (*opcional*)	Representa rectángulos con color en el plano, con lados paralelos a los ejes.	• GetBase • GetAltura • GetPerímetro • GetÁrea • GetLongitudDiagonal • IsCuadrado • GetVértice (*opcional*): Retorna el punto correspondiente a cada uno de los cuatro vértices, es decir, *SuperiorIzquierdo*, *SuperiorDerecho*, *InferiorIzquierdo*, e *InferiorDerecho*.
Polígono	Representa polígonos con color en el plano.	• AddVértice • GetVértice • SetVértice • RemoveVértice • GetCantidadLados • GetPerímetro

27.5. Restricciones

- Se debe usar el Capítulo 19, *Tipo Color* construido previamente.
- Los vértices deben ser del tipo *Punto*.
- La secuencia dinámica de vértices debe implementarse con un `array` que contenga los elementos y un `unsigned` que indique cuantos vértices tiene realmente. Ese `unsigned` es menor o igual al tamaño del `array`.
- Los vértices deben ser del tipo *Punto*.
- Las pruebas deben realizarse con `assert`, sin usar `cin` ni `cout`.

27.6. Tareas

Por cada tipo de dato:

1. Especificar el tipo matemáticamente.
2. Diseñar y codificar las pruebas en `main`.
3. Implementar el tipo.

27.7. Productos

```
DD-Geometría
 |-- readme.md
 |-- Geometría.md       // Especificación todos los tipos
 `-- Geometría.cpp      // Implementación y pruebas de todos los tipos
```

Crédito Extra

Estructurar la solución con separación física en archivos de pruebas, de implementación parte privada, y de implementación parte pública.

Escribir un `makefile` que construya y pruebe la solución. Estos temas están desarrollados en [Interfaces-Make]

```
DD-Geometría
 |-- readme.md
 |-- Makefile
 |-- Color.md            // Especi|ficación
 |-- Color.h             // Implmntcn Parte Pública
 |-- ColorTest.cpp       // Pruebas
 |-- Color.cpp           // Implmntcn Parte Privada
 |-- Punto.md            // Especificación
 |-- Punto.h             // Implmntcn Parte Pública
 |-- PuntoTest.cpp       // Pruebas
 |-- Punto.cpp           // Implmntcn Parte Privada
 |-- Círculo.md          // Especificación
 |-- Círculo.h           // Implmntcn Parte Pública
 |-- CírculoTest.cpp     // Pruebas
 |-- Círculo.cpp         // Implmntcn Parte Privada
 |-- Triángulo.md        // Especificación
 |-- Triángulo.h         // Implmntcn Parte Pública
```

```
|-- TriánguloTest.cpp   // Pruebas
|-- Triángulo.cpp       // Implmntcn Parte Privada
|-- Rectángulo.md       // Especificación
|-- Rectángulo.h        // Implmntcn Parte Pública
|-- RectánguloTest.cpp  // Pruebas
|-- Rectángulo.cpp      // Implmntcn Parte Privada
|-- Polígono.md         // Especificación
|-- Polígono.h          // Implmntcn Parte Pública
|-- PolígonoTest.cpp    // Pruebas
`-- Polígono.cpp        // Implmntcn Parte Privada
```

28

Geometría Parte II — Input/Output

28.1. Introducción

Esta trabajo es el segundo en la serie que aplica tipos para solucionar problemas de geometría, utiliza los tipos de la primera parte.

Este trabajo tiene como tema central la construcción de tipos mediante producto cartesiano;

28.2. Problema

Dado un archivo con polígonos, copiar a un nuevo archivos los polígonos que tienen un perímetro menor a un valor *x*.

28.3. Restricciones

- La conexión a los archivos debe ser mediante `stream`s.
- Si fuese necesario utilizar `in.clear()` para limpiar el estado erróneo y volver a leer de un `stream`.
- A los tipos deben agregarse operaciones de extracción e inserción según los siguientes prototipos, donde *T* es el nombre del tipo:

```
bool ExtraerT(istream& in, T& v);

bool InsertarT(ostream& out, const T& v);
```

Crédito Extra

Utilizar *interfaz fluida* [FLUENT] con estos prototipos para extracción e inserción respectivamente:

```
istream& ExtraerT(istream& in, T& v);

ostream& InsertarT(ostream& out, const T& v);
```

- La solución debe desarrollarse en una función:
  `void CopiarPolígonosConPerímetrosMayoresA(double x, string nombreArchivoIn, string nombreArchivoOut);`

28.4. Tareas

1. Diseñar la representación que cada tipo va a tener en los flujos.
2. (Opcional) Especificar matemáticamente la operación inserción para cada tipo, la especificación de la opearción extracción es simplemente: "*La operación Extraer debe poder extrar un valor insertado por la operación Insertar*".
3. Agregar las pruebas del par de operaciones para cada tipo.

28.5. Productos

Este trabajo modifica los productos del trabajo anterior.

Si decidiste hacer un solo archivo `Geometría.cpp` con su `main` invocá a `CopiarPolígonosConPerímetrosMayoresA` al final del `main`.

Si decidiste separar los archivos, creá un nuevo arhivo llamado `Filtrar.cpp` con un `main` que invoque a la función `CopiarPolígonosConPerímetrosMayoresA` y con la implementación de esa función.

29

Geometría Parte III — Estructuras Enlazadas

Esta parte resuelve el mismo problema que la anterior, la diferencia es que aplica estructuras enlazadas en vez de contiguas.

Los cambios deben estar acotados a la declaración de los structs y a la implementación de las operaciones, pero no debe cambiar su prototipo.

30

Geometría Parte IV — Renderizar

30.1. Polígonos en SVG

30.1.1. Objetivos

- Demostrar la implementación de tipos secuenciales dinámicos y aplicar lo visto en anterior trabajo.

30.1.2. Tareas

- Especificación del tipo polígono que tiene cantidad dinámica de puntos y cuenta con las operaciones *GetCantidadDePuntos*, *GetPunto*, *SetPunto*, *InsertarPunto*, *RemoverPunto*.
- Implementación *contigua*.
- Implementación *enlazada*.
- Prueba con `assert` y generación del polígono en SVG.

30.2. Productos

```
DD-PoligonosSvg
 |-- readme.md
 |-- Polígono.md           // Especificación del tipo polígono
 |-- PolígonoContigua.cpp  // Implementación y pruebas
 `-- PolígonoEnlazada.cpp  // Implementación y pruebas
```

Parte VII. Trabajos sobre Estructuras Dinámicas — Secuencias, Pilas, y Colas

Implementaciones contiguas y enlazadas.

31

Secuencia Dinámica — Implementación Contigua

31.1. Restricciones

- La implementación debe basarse en `array`, por lo tanto tienen una capacidad máxima.

31.2. Tareas

- Especificar tipo.
- Diseñar pruebas.
- Implementar parte pública.
- Implementar parte privada.
- Probar.
- Diseñar un programa de aplicación.

31.3. Productos

```
DD-SecDinCont
 |-- readme.md
 |-- SecDin.md // Especificación.
 |-- SecDinTest.cpp
 |-- SecDin.h
 |-- SecDinCont.cpp
 `-- SecDinApp.cpp
```

32

Stack — Implementación Contigua

32.1. Restricciones

- La implementación debe basarse en `array`, por lo tanto tienen una capacidad máxima.

32.2. Tareas

- Especificar tipo.
- Diseñar pruebas.
- Implementar parte pública.
- Implementar parte privada.
- Probar.
- Diseñar un programa de aplicación.

32.3. Productos

```
DD-StackCont
 |-- readme.md
 |-- Stack.md // Especificación.
 |-- StackTest.cpp
 |-- Stack.h
 |-- StackCont.cpp
 `-- StackApp.cpp
```

33

Queue — Implementación Contigua

33.1. Restricciones

- La implementación basarse en `array`, por lo tanto tienen una capacidad máxima.
- El `array` debe utilizarse como un array circular con artimética módulo N.

33.2. Tareas

- Especificar tipo.
- Diseñar pruebas.
- Implementar parte pública.
- Implementar parte privada.
- Probar.
- Diseñar un programa de aplicación.

33.3. Productos

```
DD-QueueCont
 |-- readme.md
 |-- Queue.md // Especificación.
 |-- QueueTest.cpp
 |-- Queue.h
 |-- QueueCont.cpp
 `-- QueueApp.cpp
```

34

Secuencia Dinámica — Implementación Enlazada

34.1. Restricciones

- La implementación deben basarse en una `struct` con un puntero al primer nodo.
- La reserva de memoria para los nodos debe realizarse dinámicamente con el operador `new`.

34.2. Tareas

- Especificar tipo.
- Diseñar pruebas.
- Implementar parte pública.
- Implementar parte privada.
- Probar.
- Diseñar un programa de aplicación.

34.3. Productos

```
DD-SecDinLink
 |-- readme.md
 |-- SecDin.md // Especificación.
 |-- SecDinTest.cpp
 |-- SecDin.h
```

```
|-- SecDinLink.cpp
`-- SecDinApp.cpp
```

35

Stack — Implementación Enlazada

35.1. Restricciones

- La implementación basarse en un struct con un puntero al nodo de la cima.
- La reserva de memoria para los nodos debe realizarse dinámicamente con el operador `new`.

35.2. Tareas

- Especificar tipo.
- Diseñar pruebas.
- Implementar parte pública.
- Implementar parte privada.
- Probar.
- Diseñar un programa de aplicación.

35.3. Productos

```
DD-StackCont
 |-- readme.md
 |-- Stack.md // Especificación.
 |-- StackTest.cpp
 |-- Stack.h
 |-- StackLink.cpp
 `-- StackApp.cpp
```

36

Queue — Implementación Enlazada

36.1. Restricciones

- La implementación basarse en una `struct` con un puntero al primer nodo y otro al último.
- La reserva de memoria para los nodos debe realizarse dinámicamente con el operador `new`.

36.2. Tareas

- Especificar tipo.
- Diseñar pruebas.
- Implementar parte pública.
- Implementar parte privada.
- Probar.
- Diseñar un programa de aplicación.

36.3. Productos

```
DD-QueueLink
 |-- readme.md
 |-- Queue.md // Especificación.
 |-- QueueTest.cpp
 |-- Queue.h
 |-- QueueLink.cpp
 `-- QueueApp.cpp
```

37

Árbol de Búsqueda Binaria

37.1. Objetivos

- Objetivo.
- Objetivo.
- Objetivo.

37.2. Temas

- Tema.
- Tema.
- Tema.

37.3. Problema

Problema

37.4. Restricciones

- Restricción.
- Restricción.
- Restricción.

37.5. Tareas

1. Tarea.

2. Tarea.
3. Tarea.

37.6. Productos

37.7. Temas

- Tipo de dato definido por el usuario (programador).
- Tipo Abstracto de Datos.
- Especificación.
- Implementación.
- Definición de conjunto de valores con `struct`.

37.8. Problema

Diseñar un tipo Círculo en el plano, con las operaciones:

- GetCircunferencia
- GetÁrea
- IsDentro, que dado un punto determina si está dentro de un círculo.
- Mover, que traslada el círculo a otro lugar del plano.

37.8.1. Analizar y Comparar

- Las diferentes formas matemáticas de definir círculos y las diferentes formas de representarlos en memoria.
- ¿Podría representarse con un `array` y con `using`?

37.9. Productos

```
DD-Círculo
 |-- readme.md
 |-- Círculo.md       // Especificación
 `-- Círculo.cpp      // Implementación y pruebas
```

Parte VIII. Otros Trabajos

38

Templates

38.1. Objetivos

- Matriz con cantidad y tipo de elemento parametrizado.
- Secuencia Dinámica Contigua con cantidad y tipo de elemento parametrizado.

39

Historial del Browser

39.1. Necesidad

Implementar la funcionalidad *back* y *forward* común a todos los browsers.

39.2. Restricciones sobre la Interacción

- Procesamiento línea a línea.
- Una línea puede contener B para *back*, F para *forward*, el resto de las líneas de las se las considera como *URL* destino correctas.
- Por cada línea leída, se debe enviar una línea a la salida estándar: si es una URL, se envía esa URL, si es B, se envía la anterior URL, y si es F, se envía la siguiente URL.
- El procesamiento finaliza cuando no hay más líneas.

Tabla 39.1. Ejemplo de interacción

Secuencia	Entrada	Salida
1	alfa	alfa
2	beta	beta
3	gamma	gamma
4	delta	delta
5	B	gamma
6	F	delta
7	B	gamma

Secuencia	Entrada	Salida
8	epsilon	epsilon
9	B	gamma
10	F	epsilon

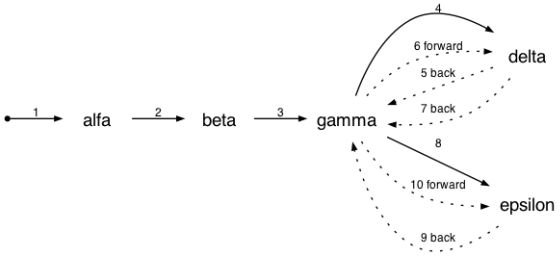


Figura 39.1. Líneas de tiempo (BTTF2) para la interacción ejemplo.

39.3. Restricciones de solución

- Obtención de líneas
  - En *C++*:

```
string línea; // guarda la línea obtenida de cin.
while(getline(cin, línea)) ... // obtiene una línea de cin y la
 guarda en línea.
```

  - En *C*:

```
#define MAX_LINE_LENGTH 1000 // cantidad máxima de caracteres en
 una línea.
char line[MAX_LINE_LENGTH+1+1]; // guarda la línea obtenida de
 stdin.
while(fgets(línea, sizeof línea, stdin)) ... // obtiene una línea
 de stdin y la guarda en línea.
```

- Diseñar las siguientes funciones:
  - `GetLínea() // retorna una línea de la entrada estándar.`
  - `GetTipo(línea) // retorna un código para los diferentes tipos de líneas.`

- `AccionarSegún( GetTipo(línea) ) // realiza la acción correspondiente.`
- `Mostrar(unaUrl) // Envía unaUrl a la salida estándar.`
- `Back() // vuelve una URL atrás y la muestra.`
- `Forward() // avanza a la URL siguiente y la muestra.`
- `GuardarUrl() // realiza lo necesario para guardar una URL.`
- `GetPrevUrl() // obtiene la anterior URL.`
- `GetNextUrl() // obtiene la sieugiente URL.`

39.3.1. Mejoras

Las siguientes mejoras son ejercicios opcionales y avanzados que completan la funcionalidad.

Nuevos Comandos para el Manejo del Historial

- `Refresh`: Envía por la salida estándar la URL actual.
- `PrintHistory`: Envía por la salida estándar todas las URL visitadas en orden, primero la primera visitada y último la la última.
- `ClearHistory`: Borra el historial.
- `PrintThisTimeLine`: Envía por la salida estándar una representación textual en *dot* [DOT] de la línea temporal actual. Para el ejemplo original, si estamos en el paso N mostraría:
- `PrintAllTimeLines`: Lo mismo que `PrintThisTimeLine` pero para todas las líneas de tiempo en forma de árbol, en vez de secuencia, cuya raíz es la primera URL visitada.
- Agregar al historial la fecha y hora de cada visita. En *C++* con `<chrono>`, y en *C* con `<time.h>`.
- Al finalizar el procesamiento, generar los archivos `History.txt`, `ThisTimeLine.gv`, y `AllTimeLines.gv`.

Mejoras al Intérprete de Comandos

- Requeerir que los comandos comiencen con `.` (punto).

- Agregar a los comandos `PrintX` una opción `-f` para indicar que la salida se envía a un file, y no a la salida estándar. Los filenames por defecto son `History.txt`, `ThisTimeLine.gv`, y `AllTimeLines.gv`, respectivamente.
- Agregar a la opción `-f` de los comandos `PrintX` un argumento para indicar el nombre del file destino, para que se puedan paersonalizar los archivos destino.
- Agregar validación de las líneas, para que el programa pueda emitir mensajes del tipo `Comando inválido.`, `Opción inválida.`, `Argumento inválido.`, y `URL inválida.`. La función que implementa la validación es `GetComandooUrl(línea)` que retorna un valor de la enumeración `{NoHayMásLíneas, Back, Forward, Url, Refresh, ClearHistory, PrintHistory, PrintThisTimeLine, PrintAllTimeLines, UrlInválida, ComandoInválido};`. Esta función de validación se puede implementar de tres formas:
  - Implementar las validaciones con las tres estructuras de control de flujo de ejecución.
  - Implementar las validaciones con un autómata finito con tantos estados finales como situaciones posibles.
  - Implementar las validaciones con expresiones regulares. En *C++* utilizar `regex`, en *C* utilizar `lex`.
- Agregar *alias* a los comandos y hacer el intérprete *case-insensitive*:

Comando	**Alias**
`Back`	`B`
`Forward`	`F`
`Refresh`	`R`
`PrintHistory`	`PH`
`ClearHistory`	`CH`
`PrintThisTimeLine`	`PTL`
`PrintAllTimeLines`	`PATL`

39.4. Productos

- `BrowserSimple/browse.cpp`
- `BrowserMásComandos/browse.cpp`
- `BrowserMejorIntérprete/browse.cpp`
- `BrowserValidadorEstructurado/browse.cpp`
- `BrowserValidadorAutómata/browse.cpp`
- `BrowserValidadorRegex/browse.cpp`

Parte IX. Back Matter

40

Bibliografía

[Git101] *Git 101* https://josemariasola.wordpress.com/papers#Git101

[CompiladoresInstalacion] *Compiladores, Editores y Entornos de Desarrollo: Instalación, Configuración y Prueba* https://josemariasola.wordpress.com/papers/#CompiladoresInstalacion

[Interfaces-Make] José María Sola. *Interfaces & Make* (2017) https://josemariasola.wordpress.com/ssl/papers#Interfaces-Make

[CharacterInputOutputRedirection] José María Sola, Jorge Muchnik. *Entrada-Salida de a Caracteres y Redirección* (2012) https://josemariasola.wordpress.com/papers/#CharacterInputOutputRedirection

[UTNORD1877] Consejo Superior de la Universidad Tecnológica Nacional *Diseño Curricular de Ingeniería en Sistemas de Información - Plan 2023* (2022) http://csu.rec.utn.edu.ar/CSU/ORD/1877.pdf

[DOT] Emden R. Gansner and Eleftherios Koutsofios and Stephen North. *Drawing graphs with dot* (2015) Retrived 2018-06-19 from https://www.graphviz.org/pdf/dotguide.pdf

[FLUENT] *Interfaz Fluida* https://en.wikipedia.org/wiki/Fluent_interface

[DAWSON2012] Bruce Dawson *Comparing Floating Point Numbers, 2012 Edition* (2012) https://randomascii.wordpress.com/2012/02/25/comparing-floating-point-numbers-2012-edition/

[ERICSON2008] Christer Ericson *Floating-point tolerances revisited* (2008) https://realtimecollisiondetection.net/blog/?p=89

[BJARNE2013] Bjarne Stroustrup *The C++ Programming Language, Fourth Edition* (2013)

[PINEIRO] María Alicia Piñeiro. *Matemática Discreta Unidad 3 Divisibilidad en Z* (2019) https://josemariasola.wordpress.com/aed/reference#gcd

[Enums] José María Sola. *Enumeraciones: Construcción de Tipo por Extensión* (2018) https://josemariasola.wordpress.com/aed/papers#Enums

[Structs-Arrays] José María Sola. *Tuplas & Secuencias y Structs & Arrays: Construcción de Tipo por Producto Cartesiano* (2018) https://josemariasola.wordpress.com/aed/papers#Structs-Arrays

41

Changelog

7.1.0+2026-06-02

- Relanzamiento trabajo *Funciones y Comparación de Valores en Punto Flotante — Celsius*.

7.0.0+2024-11-11

- Reestructuración por partes que agrupan los trabajos según grandes temáticas.
- Trabajo *Precedencia de Operadores — Bisiesto*:
  - Corrección de nombre `IsLeap` a `IsBisiesto`.
- Trabajo *Arreglos & Dimensiones — Total de Ventas*:
  - Corrección falta de archivos de prueba en los productos.
- Nuevo trabajo de Polígono en SVG.

6.1.0+2023-06-20

- Trabajo *Precedencia de Operadores — Bisiesto*:
  - Otra forma de presentar créditos extra.
  - Más créditos extra.
  - Mejoras en la redacción.

6.0.0+2023-04-30

- Nivelación con respecto a SSL sobre el trabajo #0
- Nueva forma de presentar la biblografía.

5.0.0+2021-10-10

- Nuevos trabajos:
  - Enumeraciones & Estructuras — Plano y Punto
  - Tipo Círculo
  - Tipo Triángulo
  - Estructuras & Arreglos — Cuadriláteros y Rectángulos
  - Tipo Polígono
  - Simulación de Estructuras Dinámicas — Polígonos
  - Interfaces & Implementaciones — Modularización
- Corrección de typos.

4.8.0+2021-09-07

- Nuevo trabajo *Arreglos & Dimensiones — Total de Ventas*.

4.7.0+2021-08-30

- Nuevo trabajo *Problemas, Arrays, String & Enumeraciones — CUIL*.