Trabajos de Algoritmos y Estructura de Datos

Esp. Ing. José María Sola, profesor.

Revisión 4.1.0
2020-09-03

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.
```

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`.[1]
- archivos tests.[1]

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.

[1] Para algunos trabajos, el archivo `Makefile` y los tests son obligatorios, de ser así, se indica en el enunciado del trabajo.

```
usuario // Usuario GithHub
`-- Asignatura // Repositorio personal público para a la asignatura
    |-- readme.md // Front page del usuario
    |-- 00-Hello // Carperta 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 // Carperta de resolución de trabajo
|    |-- .gitignore
|    |-- readme.md // Front page de la resolución
|    |-- Makefile
|    |-- StackTest.cpp
|    |-- Stack.h
|    |-- Stack.cpp
|    `-- StackApp.cpp
`-- 01-Otro-trabajo
```

2.1.2. Lenguaje de Programación

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

2.1.3. 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[2] se publican cuando es la fecha y hora límite de entrega de cada trabajo.

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

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 con, un programa simple, que se está en capacidad de editar, compilar, y ejecutar un programa C++.
- Contar con las herramientas necesarias para abordar la resolución de los trabajos posteriores.

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. Tareas

1. Solicitar inscripción al Grupo Yahoo, la aprobación demora un par de días.
2. Si no posee una cuenta *GitHub*, crearla.

3. Crear un repositorio público llamado `AED`.
4. Escribir el archivo `readme.md` que actúa como *front page* del repositorio personal.
5. Crear la carpeta `00-CppHelloWorld`.
6. Escribir el archivo `readme.md` que actúa como *front page* de la resolución.
7. Seleccionar, instalar, y configurar un compilador **C++ 20** (ó **C++ 17** ó **C++ 14** ó **C++ 11**).
8. Indicar en `readme.md` el compilador seleccionado.
9. Probar el compilador con un programa `hello.cpp` que envíe a `cout` la línea `Hello, World!` o similar.
10. Ejecutar el programa, y capturar su salida en un archivo `output.txt` .
11. Publicar en el repositorio personal `AED` la carpeta `00-CppHelloWorld` con `readme.md`, `hello.cpp`, y `output.txt` .
12. La última tarea es informar por email a UTNFRBAAED@yahoogroups.com[1] el usuario *GitHub*.

4.6. Productos

```
Usuario
`-- AED
    `-- 00-CppHelloWorld
        |-- readme.md
        |-- hello.cpp
        `-- output.txt
```

[1] mailto:UTNFRBAAED@yahoogroups.com

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 ???, 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

Ejemplos de Valores y Operaciones de Tipos de Datos

6.1. Objetivos

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

6.2. Temas

- Tipos de datos.
- Declaraciones.
- Variables.
- Valores.

6.3. Problema

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

6.4. Restricciones

- No extraer valores de `cin`, usar valores literales (constantes).

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 esos realmente todos los tipos que vimos en clase? Justifique.

Crédito Extra

> No utilice `cout` y sí utilice `assert` para las pruebas.

6.6. Productos

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

7

Funciones y Comparación de Valores en Punto Flotante — Celsius

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.

7.3. Problema

Desarrollar una función que, dada una magnitud en Farehnheit, calcule la equivalente en Celsius:

Hay dos sub-problemas que se requieren solucionar antes de poder probar e implementar la función `Celsius`:

- Valor de la fracción $\frac{5}{9}$ versus la división entera de la expresión `5/9` en C++.
- Representación 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 repesente la tolerancia.

7.4. Restricciones

- Las pruebas deben realizarse con `assert`.
- Los prototipos deben ser:

```
double Celsius(double);
bool AreNear(double, double, double = 0.001);
```

7.5. Tareas

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

7.6. Productos

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

8

Funciones y Operador Condicional

8.1. Objetivos

- Demostrar manejo de funciones y del operador condicional.

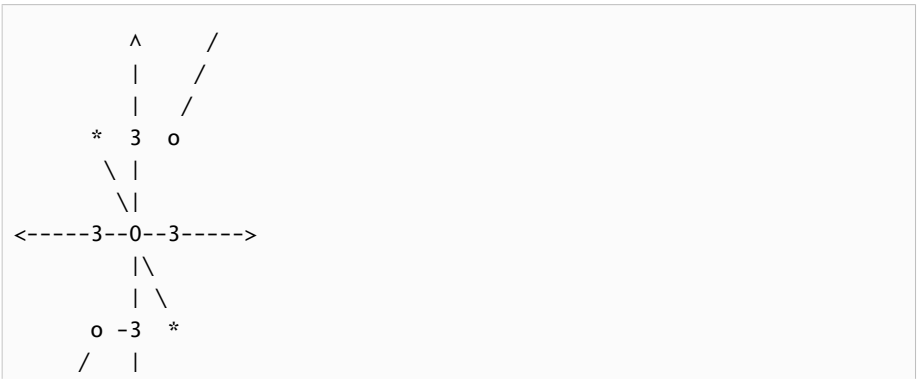
8.2. Temas

- 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.
- Efecto de lado de una expresión.
- Funciones.

9.3. Problema

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

9.4. Restricciones

- El nombre de la función debe ser `IsBisiesto` [1].
- Aplicar operadores booleanos
- No aplicar el operador condicional.
- No aplicar `if` ni `switch`.
- Las pruebas deben realizarse con `assert`.

9.5. Tareas

1. Escribir la definición matemática de la función.
2. Escribir las pruebas.
3. Escribir el prototipo.
4. Escribir la definición.
5. Incluir en `readme.md` el *árbol de expresión* asociado a la expresión de retorno de la función.

9.6. Productos

```
DD-Bisiesto
 |-- readme.md
 `-- IsBisiesto.cpp
```

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

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

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.

11.1. Productos

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

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

12

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`?

12.1. 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
```

13

Tipo Color

13.1. Objetivos

- Demostrar capacidad de construcción de tipos compuestos basados en tipos existentes atómicos.

13.2. Temas

- Tipos.
- Definición de conjunto de valores con `struct`.
- Tipos enteros de ancho fijo.
- Variables externas.
- Variables `const`.

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

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

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 incializar 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.

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

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.

13.4. 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.
- 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` e indicar porque, si no es posible usar `uint8_t`, es correcto usar `unsigned char` y no `char`.

13.5. 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.

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

14

Geometría

Estos trabajos tienen como tema central la construcción de tipos mediante producto cartesiano. El tema se desarrolla en [TUPLES_SEQS_STRUCTS_ARRAYS].

14.1. Tipo Punto

14.1.1. Objetivos

- Demostrar capacidad de construcción de tipos compuestos basados en tipos existentes atómicos.

14.1.2. Temas

- Tipos.
- Definición de conjunto de valores con `struct`.
- Definición de conjunto de operaciones con funciones y pasaje de argumentos por referencia (i.e., variable).

14.1.3. Problema

Desarrollar el tipo de dato *Punto* que representa un punto en el plano con coordenadas cartesianas. Las operaciones son: *IsIgual*, *GetDistancia*, y *GetDistanciaAlOrigen*.

Crédito Extra

> Definir las siguientes operaciones y otras que se te ocurran: *GetRho*, *GetPhi*, *GetCuadrante*, *GetEje*, *GetSemiplano*, *Mover*.

14.1.4. Restricciones

- Las pruebas deben realizarse con `assert`, sin usar `cin` ni `cout`.

14.1.5. Tareas

1. Especificar matemáticamente el tipo en `Punto.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 `Punto` 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.

14.1.6. Productos

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

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-Punto
 |-- readme.md
 |-- Makefile
 |-- Punto.md       // Especificación
 |-- Punto.h        // Implementación: Parte Pública
 |-- PuntoTest.cpp  // Pruebas
 `-- Punto.cpp      // Implementación: Parte Privada
```

14.2. Tipo Círculo

14.2.1. Objetivos

- Demostrar capacidad de construcción de tipos compuestos basados en otros tipos compuestos.

14.2.2. Temas

- Tipos.
- Definición de conjunto de valores con `struct`.
- Definición de conjunto de operaciones con funciones y pasaje de argumentos por referencia (i.e., variable).

14.2.3. Problema

Desarrollar el tipo de dato *Círculo* que representa un círculo con color en el plano. Las operaciones son: *GetCircunferencia*, *GetÁrea*, y *Mover*.

14.2.4. Restricciones

- Las pruebas deben realizarse con `assert`, sin usar `cin` ni `cout`.

14.2.5. Tareas

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

14.2.6. Productos

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

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-Círculo
 |-- readme.md
 |-- Makefile
 |-- Círculo.md       // Especificación
 |-- Círculo.h        // Implementación: Parte
Pública
 |-- CírculoTest.cpp  // Pruebas
 `-- Círculo.cpp      // Implementación: Parte
Privada
```

14.3. Tipo Triángulo

14.3.1. Objetivos

- Demostrar capacidad de construcción de tipos compuestos basados en otros tipos compuestos.

14.3.2. Temas

- Tipos.
- Definición de conjunto de valores con `struct`.
- Definición de conjunto de operaciones con funciones y pasaje de argumentos por referencia (i.e., variable).

14.3.3. Problema

Desarrollar el tipo de dato *Triángulo* que representa triángulos con color en el plano. El triangulo se lo describe por tres puntos. Las operaciones son: *GetPerímetro*, *GetÁrea*, *IsEscaleno*, *IsEquilátero*, e *IsIsósceles*.

Crédito Extra

> Definir las operaciones *GetTipo*, que retorna el tipo de un triángulo, y *GetCentro*, que retorna el punto que representa el centro del triángulo.

14.3.4. Restricciones

- Las pruebas deben realizarse con `assert`, sin usar `cin` ni `cout`.

14.3.5. Tareas

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

14.3.6. Productos

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

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-Triángulo
 |-- readme.md
 |-- Makefile
 |-- Triángulo.md       // Especificación
 |-- Triángulo.h        // Implementación: Parte
 Pública
 |-- TriánguloTest.cpp  // Pruebas
 `-- Triángulo.cpp      // Implementación: Parte
 Privada
```

14.4. Tipo Rectángulo

14.4.1. Objetivos

- Demostrar capacidad de construcción de tipos compuestos basados en otros tipos compuestos.

14.4.2. Temas

- Tipos.
- Definición de conjunto de valores con `struct`.
- Definición de conjunto de operaciones con funciones y pasaje de argumentos por referencia (i.e., variable).

14.4.3. Problema

Desarrollar el tipo de dato *Rectángulo* que representa rectángulos con color en el plano, con lados paralelos a los ejes. Las operaciones son: *GetBase*, *GetAltura*, *GetPerímetro*, *GetÁrea*, *GetLongitudDiagonal, _IsCuadrado*.

Crédito Extra

Definir la operación *GetVértice*, que retorna el punto correspondiente a cada uno de los cuatro vértices: *SuperiorIzquierdo*, *SuperiorDerecho*, *InferiorIzquierdo*, e *InferiorDerecho*.

14.4.4. Restricciones

- Las pruebas deben realizarse con `assert`, sin usar `cin` ni `cout`.

14.4.5. Tareas

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

14.4.6. Productos

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

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-Rectángulo
 |-- readme.md
 |-- Makefile
 |-- Rectángulo.md       // Especificación
 |-- Rectángulo.h        // Implementación: Parte
 Pública
```

```
|-- RectánguloTest.cpp  // Pruebas
`-- Rectángulo.cpp      // Implementación: Parte
Privada
```

14.5. Tipo Polígono

14.5.1. Objetivos

- Demostrar capacidad de construcción de tipos compuestos basados en otros tipos compuestos.

14.5.2. Temas

- Tipos.
- Definición de conjunto de valores con `struct`.
- Definición de conjunto de operaciones con funciones y pasaje de argumentos por referencia (i.e., variable).

14.5.3. Problema

Desarrollar el tipo de dato *Polígono* que representa polígonos con color en el plano. Las operaciones son: *AddVértice*, *GetVértice*, *SetVértice*, *RemoveVértice*, *GetCantidadLados*, y *Get_GetPerímetro*.

14.5.4. Restricciones

- Las pruebas deben realizarse con `assert`, sin usar `cin` ni `cout`.

14.5.5. Tareas

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

14.5.6. Productos

```
DD-Polígono
```

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

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-Polígono
 |-- readme.md
 |-- Makefile
 |-- Polígono.md       // Especificación
 |-- Polígono.h        // Implementación: Parte
Pública
 |-- PolígonoTest.cpp  // Pruebas
 `-- Polígono.cpp      // Implementación: Parte
Privada
```

15

Diagonal de una Matriz

15.1. Objetivos

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

15.2. Restricciones

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

15.3. Productos

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

16

Secuencia Dinámica — Implementación Contigua

16.1. Restricciones

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

16.2. Tareas

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

16.3. Productos

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

17

Templates

17.1. Objetivos

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

18

Stack — Implementación Contigua

18.1. Restricciones

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

18.2. Tareas

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

18.3. Productos

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

19

Queue — Implementación Contigua

19.1. 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.

19.2. Tareas

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

19.3. Productos

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

20

Secuencia Dinámica — Implementación Enlazada

20.1. 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`.

20.2. Tareas

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

20.3. Productos

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

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

21

Stack — Implementación Enlazada

21.1. 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`.

21.2. Tareas

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

21.3. Productos

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

22

Queue — Implementación Enlazada

22.1. 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`.

22.2. Tareas

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

22.3. Productos

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

23

Árbol de Búsqueda Binaria

23.1. Objetivos

- Objetivo.
- Objetivo.
- Objetivo.

23.2. Temas

- Tema.
- Tema.
- Tema.

23.3. Problema

Problema

23.4. Restricciones

- Restricción.
- Restricción.
- Restricción.

23.5. Tareas

1. Tarea.

2. Tarea.
3. Tarea.

23.6. Productos

24

Repetición

25

Mayor de dos Números

25.1. Problema

Dado dos números informar cuál es el mayor.

25.2. Productos

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

26

Repetición de Frase

26.1. Problema

Enviar una frase a la salida estándar muchas veces.

26.2. Restricciones

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

- Salto condicional.
- Iterativa esctructurada.

26.3. Productos

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

26.4. Entrega

- Abr 27, 13hs.

27

? Trabajo #5 — Especificación del Tipo de Dato Fecha

27.1. Tarea

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

27.2. Productos

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

28

Trabajo #9 — Browser

28.1. Necesidad

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

28.2. 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 28.1. 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

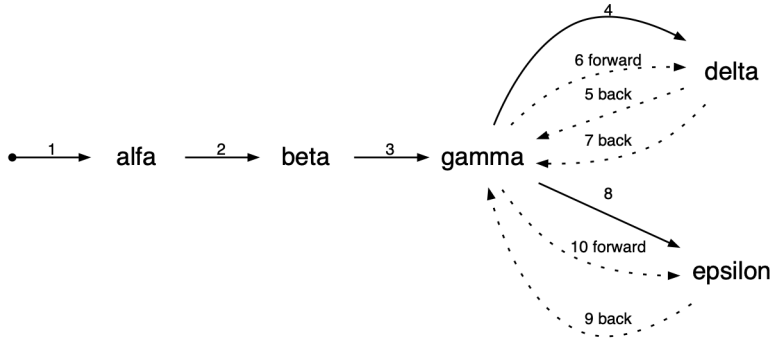


Figura 28.1. Líneas de tiempo (BTTF2) para la interacción ejemplo.

28.3. 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.`

28.3.1. 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`

Comando	Alias
PrintAllTimeLines	PATL

28.4. Productos

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

Bibliografía

Gansner, Emden R., Eleftherios Koutsofios, and Stephen North. "Drawing graphs with dot." (2015). http://graphviz.org/doc/dotguide.pdf

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

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

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