

Luis Rodrigo Morales Florián

202208521

LABORATORIO ESTRUCTURAS DE DATOS Sección B

Fase 1

---

# Manual Técnico

---

## Descripción del proyecto

La empresa "Pixel Print Studio" se dedica a la impresión de figuras de pixel art en distintos tamaños y tipos de papel. Con el crecimiento constante de clientes, ha surgido la necesidad de optimizar los procesos de recepción y producción. En este sentido, se solicita a un estudiante de ingeniería en sistemas aplicar conocimientos en estructuras de datos para mejorar la eficiencia operativa.

Deberás desarrollar una aplicación que utilice estructuras de datos y algoritmos para simular los diversos procesos en la empresa "Pixel Print Studio". La aplicación deberá representar visualmente las estructuras mediante bibliotecas compatibles, como Graphviz.

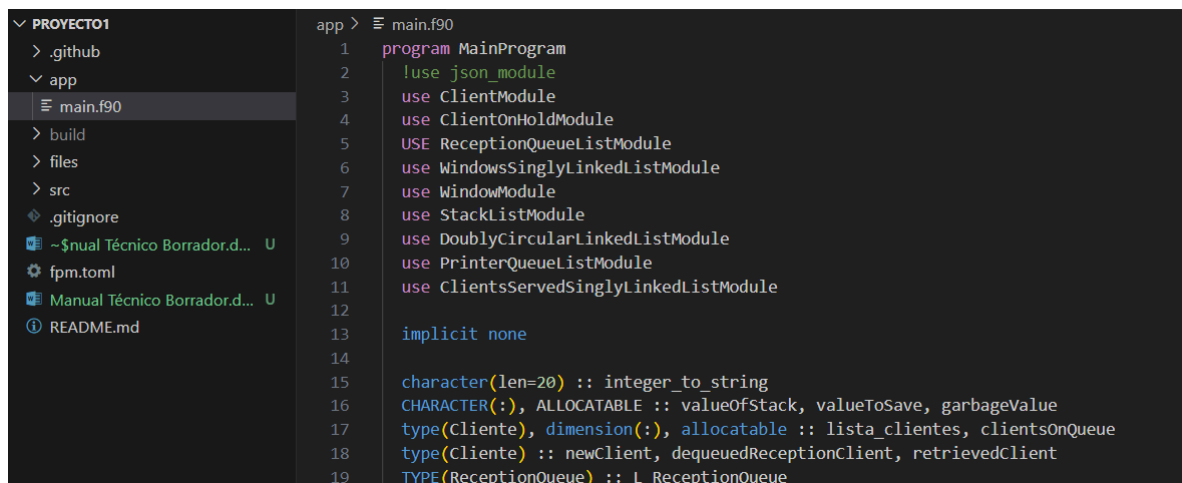
## Generalidades

- El manejador de proyectos y paquetes es FPM (Fortran Package Manager)
- Para la graficación cuenta con la generación de un archivo de texto, el cual genera un lenguaje entendible para Graphviz, de modo que luego se tendrá que copiar ese código y ponerlo en un intérprete para poder observar el resultado

- Se realiza carga masiva de clientes a través de un json, y la librería que procesa los JSON se llama json\_module
- Se utiliza un sistema de:
  - o Main File: El archivo principal donde arranca la aplicación, aquí mismo se programan y ejecutan los procesos de cada módulo de la tienda.
  - o Models: Son modelos de datos, los cuales se requieren para una mejor estructura. Son modules de fortran, que gracias a sus atributos, éstos modelos pueden ser implementados en otros lados y hacer uso de sus atributos e incluso hasta métodos
  - o Lists: Son todas las listas TDA que utilizamos en el proyecto. En mi caso, opté por crear algunas listas “personalizadas”, de modo que el nodo de cada lista aceptara diferentes tipos de datos, ya sean nativos o de los Models.

Se procede entonces con la descripción de cada parte del sistema:

## Main File



```

app > main.f90
1  program MainProgram
2      !use json_module
3      use ClientModule
4      use ClientOnHoldModule
5      use ReceptionQueueListModule
6      use WindowsSinglyLinkedListModule
7      use WindowModule
8      use StackListModule
9      use DoublyCircularLinkedListModule
10     use PrinterQueueListModule
11     use ClientsServedSinglyLinkedListModule
12
13     implicit none
14
15     character(len=20) :: integer_to_string
16     CHARACTER(:), ALLOCATABLE :: valueOfStack, valueToSave, garbageValue
17     type(Cliente), dimension(:), allocatable :: lista_clientes, clientsOnQueue
18     type(Cliente) :: newClient, dequeuedReceptionClient, retrievedClient
19     TYPE(ReceptionQueue) :: L_ReceptionQueue
  
```

Básicamente se inicia con la declaración de variables con cada uno de sus respectivos tipos. Todas las variables en pantalla:

```

character(len=20) :: integer_to_string
CHARACTER(:, ALLOCATABLE :: valueOfStack, valueToSave, garbageValue
type(Cliente), dimension(:), allocatable :: lista_clientes, clientsOnQueue
type(Cliente) :: newClient, dequeuedReceptionClient, retrievedClient
TYPE(ReceptionQueue) :: L_ReceptionQueue
type(WindowsSinglyLinkedList) :: L_Windows
type(DoublyCircularLinkedList) :: L_ClientsOnHold
type(Window) :: window1, window2, toSaveWindow, retrievedWindow
type(ClientOnHold) :: clientOnHoldToSave, clientOnHoldFromList
type(PrinterQueueList) :: L_BigPrinter, L_SmallPrinter
type(ClientsServedSinglyLinkedList) :: L_ClientsServed

integer :: opcion, subopcion, num_windows, i, step, string_to_integer, window1_img_p, window1_img_g, window2_img_p, window2_img_g
integer :: string_to_integer_2, dll_length, j

```

Luego se procede con la ejecución del menú, donde cada opción está separada por su propia subrutina:

```

DO
CALL MostrarMenu()

READ(*, *) opcion

SELECT CASE (opcion)
CASE (1)
DO
CALL MostrarSubmenu1()

READ(*, *) subopcion

SELECT CASE (subopcion)
CASE (1)
CALL Option1_1()
CASE (2)
CALL Option1_2()
CASE (3)
EXIT
CASE DEFAULT
PRINT *, "Opcion no valida, por favor intenta de nuevo."
END SELECT
END DO
CASE (2)
call Option2()
CASE (3)
PRINT *, "Elegiste la Opción 3"
CASE (4)
PRINT *, "Elegiste la Opción 4"
CASE (5)
PRINT *, "Elegiste la Opción 5"
CASE (6)
PRINT *, "Has salido del programa."
EXIT

```

Hablaremos de dos destacables con el objetivo de ejemplificar un poco, las demás opciones usted tome en cuenta que son similares, así que estos ejemplos deberían de ser suficientes.

## 1.2. Cantidad de ventanillas

```
SUBROUTINE Option1_2()
  PRINT *
  PRINT *
  PRINT *, "HAS ELEGIDO: '1.2. Cantidad de ventanillas'"
  PRINT *
  PRINT *, "Escribe la cantidad de ventanillas que deseas establecer:"
  READ(*, *) num_windows

  IF(num_windows < 1) THEN
    PRINT *, "ERROR: El numero de ventanillas debe de ser mayor a 0"
  ELSE
    write(*, '(A, I0)') "Has establecido como numero de ventanillas: ", num_windows
    DO i=1, num_windows
      CALL initializeWindow(toSaveWindow)
      CALL WindowsSinglyInsertAtEnd(L_Windows, toSaveWindow)
    END DO
  END IF
END SUBROUTINE Option1_2
```

Aquí se asignan las ventanillas que el cliente requiera, luego se crea una lista simplemente enlazada con esa cantidad de ventanillas. El número se queda guardado en el sistema y cada vez que se quiera tratar con las ventanillas, se procede con una iteración DO de  $i=1$  hasta `num_windows`, que es la cantidad establecida por el usuario.

## 2. Ejecutar paso

```
SUBROUTINE Option2()
PRINT *
PRINT *
PRINT *, "HAS ELEGIDO: '2. Ejecutar paso'"
PRINT *

IF(num_windows == -1) THEN
PRINT *, "ERROR: Aun no has asignado el numero de ventanillas"
ELSE
WRITE(integer_to_string, '(I0)') step

PRINT *, "(", TRIM(integer_to_string), ")"
PRINT *, "Acciones que se realizaron en el paso ", TRIM(integer_to_string), ":"

! Método: Por cada cliente en espera, caminar su "paso" y si ya pasaron los pasos para cada impresión, pasarlos
i = 1
DO WHILE (DoublyCircularLinkedListNodeExistsAtPosition(L_ClientsOnHold, i))
clientOnHoldFromList = DoublyCircularLinkedListGetAtPosition(L_ClientsOnHold, i)
retrievedClient = clientOnHoldFromList%client

IF(clientOnHoldFromList%img_g_done .AND. clientOnHoldFromList%img_p_done) THEN
CALL DoublyCircularLinkedListDeleteAtPosition(L_ClientsOnHold, i)
CALL ClientsServedSinglyInsertAtEnd(L_ClientsServed, retrievedClient)
PRINT *, " - El cliente con ID: ",retrievedClient%id, " termina todos sus procesos y se da por atendido."
ELSE
IF(clientOnHoldFromList%waitStep == 1) then
READ(retrievedClient%img_p, *) string_to_integer
DO j=1, string_to_integer
garbageValue = DequeuePrinter(L_SmallPrinter)
valueToSave = "IMG P"
CALL ImagesSinglyInsertAtEnd(clientOnHoldFromList%imagesLinkedList, valueToSave)
END DO

CALL DoublyCircularLinkedListIncrementWaitStepAtPosition(L_ClientsOnHold, i)
IF(string_to_integer > 0) THEN
```

Aquí podemos observar que, por cada módulo del programa, se tiene un bloque donde se hacen los chequeos respectivos y finalmente, se procede con los pasos que se requieran.

Básicamente es así como está distribuido el main file.

## Models

```
module WindowModule
  USE StackListModule
  use ClientModule

  implicit none

  type Window
    type(Cliente) :: windowClient
    type(StackList) :: imagesStack
  end type Window

  CONTAINS
  subroutine initializeWindow(windowData)
    type(Window), INTENT(OUT) :: windowData

    CALL InitializeClient(windowData%windowClient)
    call InitializeStackList(windowData%imagesStack)
  end subroutine initializeWindow
end module WindowModule
```

Como se había dicho, cuando se necesita generar una estructura específica en el programa, se recurre a módulos, y se aprovecha sus capacidades de generar atributos. De esta manera, se puede generar tipos de datos. Recordamos que para acceder a un atributo de una variable del tipo Modelo es con `modelo%atributo`, lo cual hace más eficiente el manejo de datos en fortran.

Al no existir algo como constructores, ya que fortran no es un lenguaje que tenga muy “pulida” la programación orientada a objetos, se recurre a una subrutina, donde se envía una variable del tipo del Modelo y se le agrega lo que se requiera.

## Lists

```
MODULE StackListModule
  IMPLICIT NONE

  TYPE StackNode
    CHARACTER(:), ALLOCATABLE :: nodeValue
    TYPE(StackNode), POINTER :: nextNode
  END TYPE StackNode

  TYPE StackList
    TYPE(StackNode), POINTER :: topNode
  END TYPE StackList

CONTAINS

  SUBROUTINE InitializeStackList(inputStackList)
    TYPE(StackList), INTENT(OUT) :: inputStackList
    inputStackList%topNode => NULL()
  END SUBROUTINE InitializeStackList

  SUBROUTINE StackPush(inputStackList, newValue)
    TYPE(StackList), INTENT(INOUT) :: inputStackList
    CHARACTER(:), ALLOCATABLE, INTENT(IN) :: newValue
    TYPE(StackNode), POINTER :: newNode

    ALLOCATE(newNode)
    ALLOCATE(CHARACTER(LEN(newValue)) :: newNode%nodeValue)
    newNode%nodeValue = newValue
    newNode%nextNode => inputStackList%topNode

    inputStackList%topNode => newNode
  END SUBROUTINE StackPush

  FUNCTION StackPop(inputStackList) RESULT(poppedValue)
    TYPE(StackList), INTENT(INOUT) :: inputStackList
    CHARACTER(:), ALLOCATABLE :: poppedValue
    TYPE(StackNode), POINTER :: tempNode
```

Para las listas TDA, se requiere también del uso de módulos. Aquí, gracias a la capacidad de tener atributos-métodos, y el manejo de apuntadores, es posible lograr este tipo de listas. Como decía, recurrí a crear listas personalizadas para cada lista requerida en el proyecto, por tanto, el nodo de la lista cambia a, por ejemplo, a tener uno o varios atributos de datos nativos o Modelos. Así mismo, se



crean métodos personalizados, donde también se puede acceder a una lista enviada bajo tal lista y su tipo y se puede gestionar desde ahí.

```
MODULE WindowsSinglyLinkedListModule
  use WindowModule
  IMPLICIT NONE

  TYPE Node
    type(Window) :: value
    TYPE(Node), POINTER :: next
  END TYPE Node

  TYPE :: WindowsSinglyLinkedList
    TYPE(Node), POINTER :: head
  END TYPE WindowsSinglyLinkedList

CONTAINS

  SUBROUTINE WindowsSinglyInitializeList(list)
    TYPE(WindowsSinglyLinkedList), INTENT(OUT) :: list
    list%head => NULL()
  END SUBROUTINE WindowsSinglyInitializeList

  SUBROUTINE WindowsSinglyInsertAtEnd(list, newValue)
    TYPE(WindowsSinglyLinkedList), INTENT(INOUT) :: list
    type(Window), INTENT(INOUT) :: newValue
    TYPE(Node), POINTER :: newNode, current

    call initializeWindow(newValue)

    ALLOCATE(newNode)
    newNode%value = newValue
    newNode%next => NULL()

    IF (ASSOCIATED(list%head)) THEN
      current => list%head
      DO WHILE (ASSOCIATED(current%next))
        current => current%next
      END DO
      current%next => newNode
    ELSE
      list%head => newNode
    END IF
  END SUBROUTINE WindowsSinglyInsertAtEnd
```

Aquí se puede una lista simplemente enlazada modificada para manejar todas las ventanillas. Nótese como cambia el nodo de la lista, usando tipos de Modelos fabricados por el desarrollador y también los atributos propios de la lista.