UNIVERSIDAD DE SAN CARLOS DE GUATEMALA

MANUAL TECNICO

ESTRUCTURA DE DATOS

FACULTAD DE INGENIERIA

GUATEMELA, FEBRERO 2024

INDICE

INTRODUCCIÓN

Se utilizó el lenguaje de programación Fortran para implementar y gestionar eficientemente las estructuras de datos lineales, aprovechando las características específicas del lenguaje en el desarrollo de la aplicación, la cual consiste en una imprenta de pixel arts. La aplicación representa visualmente las estructuras mediante biblioteca Graphviz.

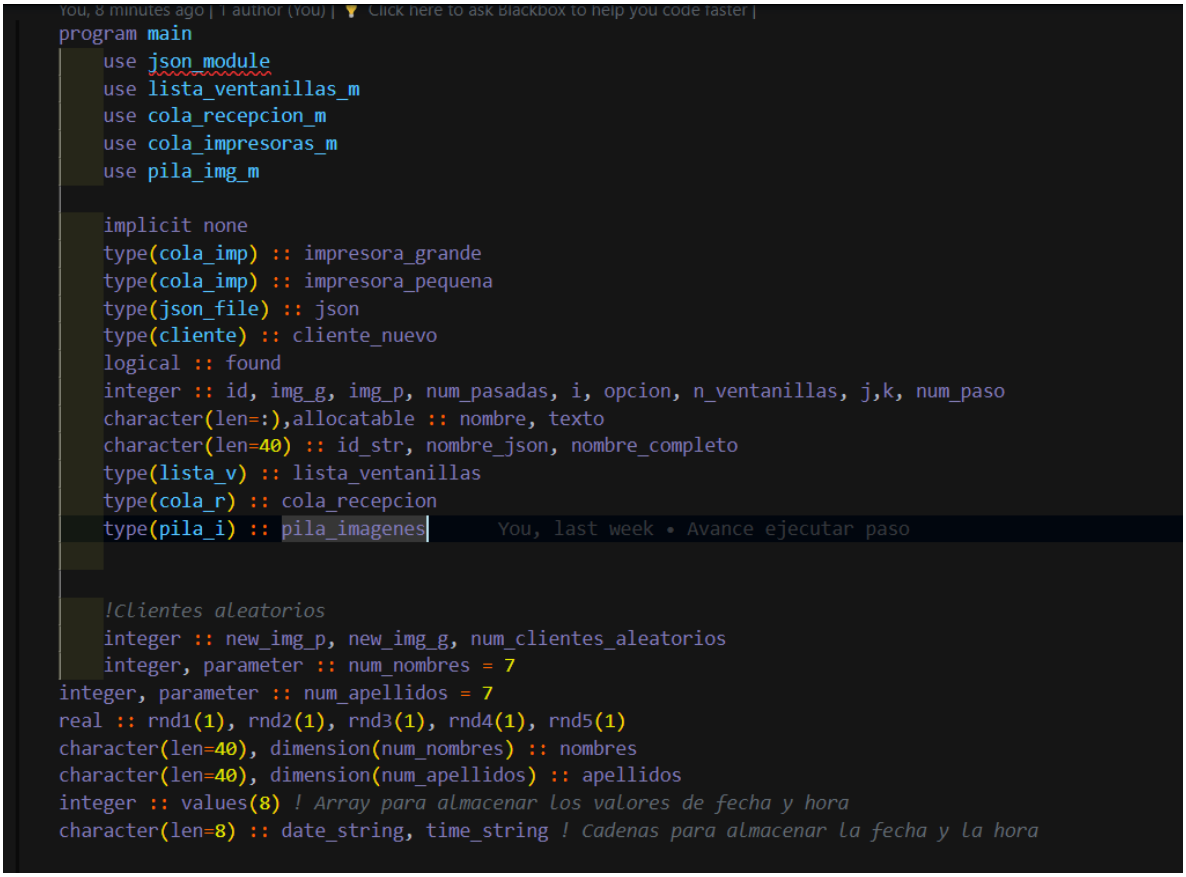
ESPECIFICACIONES DEL SISTEMA

El programa fue desarrollado en el lenguaje Fortran y su manejador de programas fpm para facilitar el orden dentro de la compilación. Además, integrar la herramienta Graphviz para generar representaciones visuales claras y comprensibles de las estructuras de datos lineales utilizadas en la simulación, facilitando así la comprensión del funcionamiento interno de la aplicación.

Guía del programa

Main.f90:

Declaraciones de módulos y variables: El programa comienza con declaraciones de módulos, que incluyen módulos definidos en otros archivos como json_module, lista_ventanillas_m, cola_recepcion_m, y cola_impresoras_m. Además, hay declaraciones de variables como impresora_grande, impresora_pequena, json, cliente_nuevo, entre otras.



Inicialización y configuración: Se inicializan las impresoras, se asignan nombres y apellidos a las variables nombres y apellidos, respectivamente. Luego, se llama a la función date_and_time para obtener la fecha y hora actual, y se inicializa el generador de números aleatorios.

```
!asignacion impresoras
call impresora_grande%nueva_impresora(1, "Impresora Grande ", 2)
call impresora_pequena%nueva_impresora(2, "Impresora Pequena", 1)
! Asignar nombres
nombres = [ "Jorge ", "Jose  ", "Juan  ", "Maria ", "Carlos", "Luis  ", "Ana   " ]

! Asignar apellidos
apellidos = ["De Leon  ", "Batres   ", "Gonzalez ", "Lopez    ", "Martinez ", "Perez    ", "Sanchez  " ]
num_paso = 1
! Llamar a date_and_time para obtener la fecha y la hora actual
call date_and_time(values=values)

! Inicializar generador de números aleatorios con el tiempo actual
call srand(seed=int(40*values(7))) ! Usamos los segundos como semilla
call json%initialize()
```

Bucle principal: Se inicia un bucle principal con la instrucción do. Dentro de este bucle, se muestra un menú principal y se lee la opción seleccionada por el usuario.

```
do
    call mostrar_menu_principal()
    read(*,*) opcion

    select case(opcion)
        case(1)
            call submenu_parametros()
        case(2)
            call ejecutar_paso()
        case(3)
            call estado_memoria()
        case(4)
            call generar_reportes()
        case(5)
            call acerca_de()
        case(6)
            exit
        case default
            print *, 'Opcion no valida. Por favor, seleccione una opcion valida.'
    end select
end do
```

Limpieza y finalización: Después de salir del bucle principal, se destruye el objeto JSON y se verifica si hubo errores en la manipulación del JSON.

```
! Limpiar
call json%destroy()
if (json%failed()) stop 4
```

Subrutinas y funciones: El programa define varias subrutinas que realizan tareas específicas, como mostrar menús, cargar clientes masivamente desde un archivo JSON, manejar la cantidad de ventanillas, ejecutar un paso de procesamiento, mostrar el estado de la memoria, generar reportes y mostrar información sobre el programa.

```
subroutine mostrar_menu_principal()
      print *, '--- Menu ---'
      print *, '1. Parametros iniciales'
      print *, '2. Ejecutar paso'
      print *, '3. Estado en memoria de las estructuras'
      print *, '4. Reportes'
      print *, '5. Acerca de'
      print *, '6. Salir'
      print *, 'Ingrese su opcion:'
end subroutine mostrar_menu_principal

subroutine submenu_parametros()
    implicit none
    integer :: opcion_parametros

    do
        call mostrar_submenu_parametros()
        read(*,*) opcion_parametros

        select case(opcion_parametros)
            case(1)
                call carga_masiva_clientes()
            case(2)
                call cantidad_ventanillas()
            case(3)
```

```
    subroutine mostrar_submenu_parametros()
          print *, '--- Parametros iniciales ---'
          print *, '1. Carga masiva de clientes'
          print *, '2. Cantidad de ventanillas'
          print *, '3. Volver al menu principal'
          print *, 'Ingrese su opcion:'
    end subroutine mostrar_submenu_parametros

    subroutine carga_masiva_clientes()
          print *, 'Ha seleccionado Carga masiva de clientes'
         ! read the file
    print *, 'Ingrese el nombre del  archivo JSON:'
    read(*, '(A)') nombre_json
call json%load(filename = nombre_json)
```

```
    subroutine ejecutar_paso()


       integer :: nuevo_id, nuevo_img_g, nuevo_img_p
       print *, 'Ha seleccionado Ejecutar paso'
       WRITE(*, '(A, I0, A)') ' ------------------------ Paso ', num_paso, ' ------------------------'
       num_paso = num_paso + 1
       call random_number(rnd5)
       num_clientes_aleatorios = mod(int(rnd5(1) * 400), 4)
       print *, 'Numero ventanillas', n_ventanillas
       ! Buscar ventanillas disponibles
       do i = 1, num_clientes_aleatorios
          call cliente_aleatorio()
       end do

       do i = 1, n_ventanillas
          if (lista_ventanillas%tiene_cliente(i)) then
              print *, 'Ventanilla', i, 'tiene cliente'

          else
              print *, 'Ventanilla', i, 'no tiene cliente'
              nuevo_id = cola_recepcion%toma_id()
              nuevo_img_g = cola_recepcion%toma_img_g()
              nuevo_img_p = cola_recepcion%toma_img_p()
              call cliente_nuevo%crear_cliente(nuevo_id, nuevo_img_g, nuevo_img_p)
              call lista_ventanillas%agregar_cliente(cliente_nuevo,i)
              call cola_recepcion%pop()
              exit !sale del samsara


          end if
       end do
       call lista_ventanillas%atender_cliente()
       call cola_recepcion%print()
       call lista_ventanillas%print_ven()
    end subroutine ejecutar_paso
```

```
    subroutine estado_memoria()
        print *, 'Ha seleccionado Estado en memoria de las estructuras'


        call impresora_grande%generar_grafo_cola_imp("cola_impresora_grande")
        call impresora_pequena%generar_grafo_cola_imp("cola_impresora_pequeña")
        call lista_ventanillas%graficar_ventanilla("ventanillas")
    end subroutine estado_memoria

    subroutine generar_reportes()
      print *, 'Ha seleccionado Reportes'
    ! Aqui puedes incluir el codigo para generar reportes
    end subroutine generar_reportes

    subroutine acerca_de()
      print *, 'Ha seleccionado Acerca de'
      print *, 'Jorge Alejandro De Leon Batres - 202111277'
    end subroutine acerca_de
end program main
```

Módulos:

pila_img.f90

Declaraciones y tipos: El módulo comienza con la declaración de un tipo node, que representa un nodo en la pila. Cada nodo contiene un id, un tipo y un enlace (next) al siguiente nodo en la pila.

```
You, 36 minutes ago | 1 author (You) | 💡 Click here to ask Blackbox to help you code faster |
module pila_img_m
    implicit none
    private

    type :: node
        private
        integer :: id !id del cliente
        character(len=5) :: tipo !img_p o img_g
        type(node), pointer :: next => null()
    end type node
```

Definición de la pila: El tipo pila_i contiene un puntero al primer nodo de la pila, denominado head.

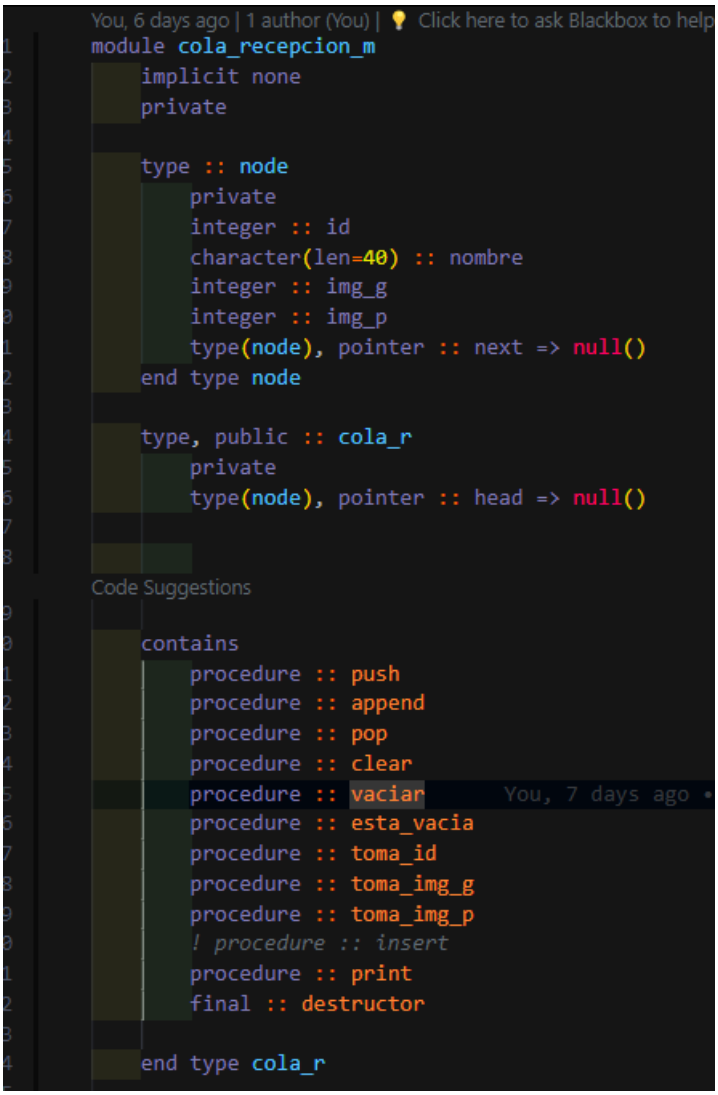
```
    type, public :: pila_i
        private
        type(node), pointer :: head => null()

    contains
        procedure :: push_i
        procedure :: get_elements
        procedure :: vaciar_i
        procedure :: pop_i
        procedure :: print_i
        final :: destructor_i
        ! Método para obtener la cabeza de la pila
        procedure :: get_head
    end type pila_i
```

Métodos y subrutinas:
push_i: Agrega un nuevo nodo a la parte superior de la pila.
pop_i: Elimina el nodo en la parte superior de la pila.
get_elements: Devuelve una cadena de caracteres que representa los elementos en la pila.
vaciar_i: Vacía la pila, eliminando todos los nodos.
print_i: Imprime los elementos de la pila.
destructor_i: Libera la memoria asignada a todos los nodos de la pila.

```
    contains

    subroutine get_head(self, head)
        class(pila_i), intent(in) :: self
        type(node), pointer :: head

        head => self%head
    end subroutine get_head
    function get_elements(self) result(elements)
        class(pila_i), intent(in) :: self
        character(len=:), allocatable :: elements
        type(node), pointer :: current

        elements = '' ! Inicializar la cadena de elementos

        current => self%head

        ! Recorrer la pila y construir la cadena de elementos
        do while (associated(current))
            elements = elements // current%tipo // ', '  !Agregar el tipo de imagen
            current => current%next
        end do

        ! Eliminar la última coma y el espacio en blanco
        if (len_trim(elements) > 0) then
            elements = elements(1:len(elements)-2)
        endif

        ! Devolver la cadena de elementos
        return
    end function get_elements
```

cola_recepcion_m

Declaraciones y tipos:
Se define un tipo node que representa un nodo en la cola de recepción. Cada nodo contiene un identificador (id), un nombre, y las cantidades de imágenes img_g e img_p, además de un enlace al siguiente nodo en la cola.
Se declara el tipo cola_r que representa la cola de recepción. Esta contiene un puntero al primer nodo de la cola (head).



Subrutinas y funciones:

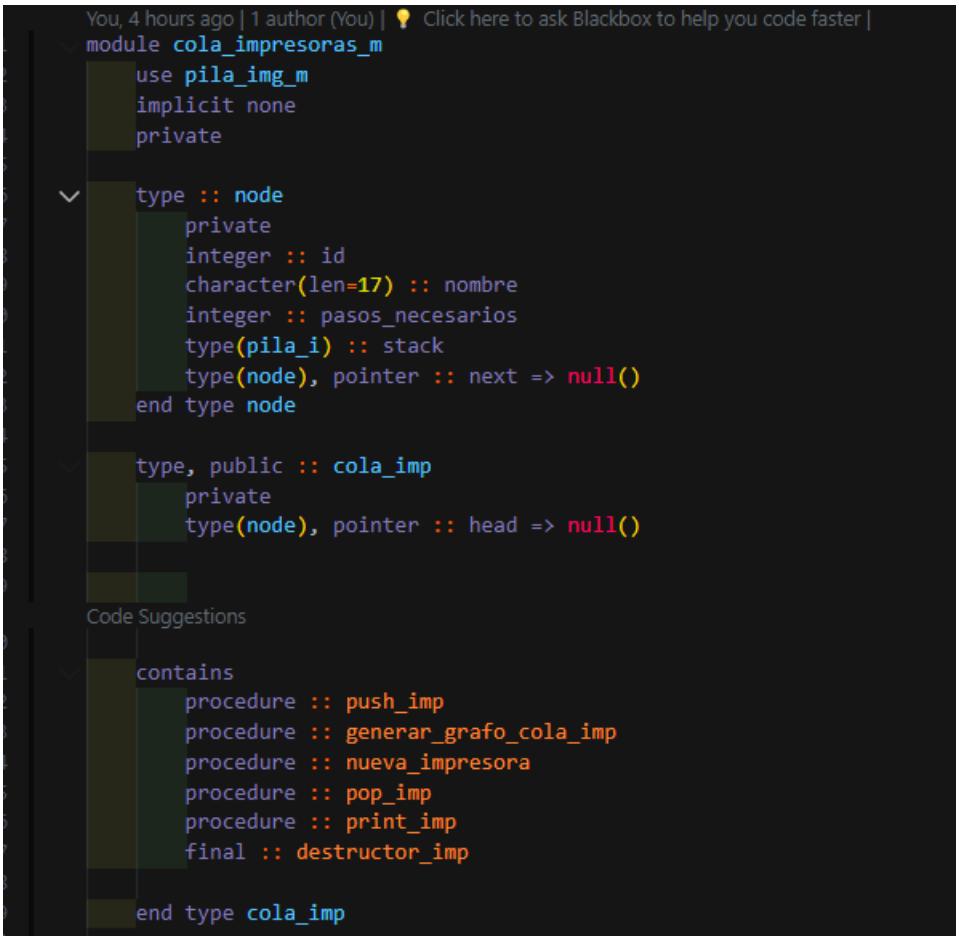
push: Agrega un nuevo nodo al final de la cola.
append: Similar a push, pero usa una interfaz diferente.
pop: Elimina el primer nodo de la cola.
clear: Elimina todos los nodos de la cola.
vaciar: Otra versión de clear con la misma funcionalidad.
esta_vacia: Verifica si la cola está vacía.
toma_id, toma_img_g, toma_img_p: Devuelven respectivamente el id, img_g e img_p del primer nodo de la cola.
print: Imprime los elementos de la cola, incluyendo el id, el nombre, img_g e img_p de cada nodo.
destructor: Libera la memoria asignada a todos los nodos de la cola.

```
function toma_id(self) result(id)
    class(cola_r), intent(inout) :: self
    type(node), pointer :: current
    integer ::id
    current => self%head
    id = current%id
end function toma_id

function toma_img_g(self) result(img_g)
    class(cola_r), intent(inout) :: self
    type(node), pointer :: current
    integer ::img_g
    current => self%head
    img_g = current%img_g
end function toma_img_g

function toma_img_p(self) result(img_p)
    class(cola_r), intent(inout) :: self
    type(node), pointer :: current
    integer ::img_p
    current => self%head
    img_p = current%img_p
end function toma_img_p

subroutine vaciar(self)
    class(cola_r), intent(inout) :: self
    type(node), pointer :: current
    type(node), pointer :: temp

    current => self%head
    do while(associated(current))
        temp => current%next
        deallocate(current)
        current => temp
    end do
    self%head => null() ! Asignar el puntero de la cabeza a nulo después de vaciar la lista
end subroutine vaciar
```

cola_impresoras_m

Declaraciones y tipos:

Se define un tipo node que representa un nodo en la cola de impresoras. Cada nodo contiene un id, un nombre, la cantidad de pasos_necesarios y una pila de imágenes representada por el tipo pila_i.
Se declara el tipo cola_imp que representa la cola de impresoras. Esta contiene un puntero al primer nodo de la cola (head).



Subrutinas:

push_imp: Agrega un nuevo nodo al final de la cola de impresoras, con un id, un nombre y un tipo_img que se insertan en la pila asociada al nodo.
generar_grafo_cola_imp: Genera un gráfico de la cola de impresoras utilizando el formato DOT y lo convierte en una imagen PNG.
nueva_impresora: Agrega una nueva impresora a la cola con un id, un nombre y la cantidad de pasos necesarios para imprimir.
pop_imp: Elimina el primer nodo de la cola de impresoras.
print_imp: Imprime los elementos de la cola de impresoras, incluyendo el id y el nombre de cada nodo, así como los elementos de la pila asociada a cada nodo.
destructor_imp: Libera la memoria asignada a todos los nodos de la cola de impresoras.

```
subroutine generar_grafo_cola_imp(self, nombre_archivo)
    implicit none
    class(cola_imp), intent(in) :: self
    character(len=*), intent(in) :: nombre_archivo
    integer :: io

    integer :: index
    character(len=100), allocatable :: command
    character(:), allocatable :: connections
    character(:), allocatable :: firsts
    character(len=8) :: name
    type(node), pointer :: current

    current => self%head
    command = "dot -Tpng " // trim(nombre_archivo) // " -o " // trim(nombre_archivo) // ".png"
    io = 1
    index = 0

    connections = ""
    firsts = ""

    open(newunit=io, file=trim(nombre_archivo))
    write(io, *) "digraph ColaImpresoras {"
    write(io, *) "  rankdir=LR;"

    if(associated(self%head)) then
        do while(associated(current))
            write(name, '(I5)') current%id

            if(firsts == "") then
                firsts = trim(name)
            end if

            if(.not. associated(current%next)) then
                connections = connections // '"' // trim(current%nombre) // '"'
                exit
            end if

            write(io, *) '"nodo' // trim(name) // '" [label="ID: ' // trim(name) // '\nNombre: ' // trim(current%nombre) // '"];'
            connections = connections // '"nodo' // trim(name) // '" -> '
    subroutine generar_grafo_cola_imp(self, nombre_archivo)
        if(associated(self%head)) then
            do while(associated(current))
                end if

                write(io, *) '"nodo' // trim(name) // '" [label="ID: ' // trim(name) // '\nNombre: ' // trim(current%nombre) // '"];'
                connections = connections // '"nodo' // trim(name) // '" -> '
                current => current%next
                index = index + 1
            end do
        end if

        write(io, *) connections
        write(io, *) "}"
        close(io)

        call execute_command_line(command, exitstat=io)

        if(io /= 0) then
            print *, "Ocurrió un error al generar la imagen."
        else
            print *, "Imagen generada satisfactoriamente."
        end if
    end subroutine generar_grafo_cola_imp

        subroutine nueva_impresora(self, id_impresora, nombre, pasos)
            implicit none
            class(cola_imp), intent(inout) :: self
            integer, intent(in) :: id_impresora
            integer, intent(in) :: pasos
            character(len=17) :: nombre
            type(node), pointer :: aux, nuevo      You, 6 days ago • Añadido las colas para las impresoras
            allocate(nuevo)
            nuevo%id = id_impresora
            nuevo%nombre = nombre
            nuevo%pasos_necesarios = pasos
            nuevo%next => null()
            if(associated(self%head)) then
                aux => self%head
                do while(associated(aux%next))
```

```
subroutine push_imp(self, id, nombre, tipo_img)
    class(cola_imp), intent(inout) :: self
    integer :: id
    character(len=17) :: nombre
    character(len=5) :: tipo_img

    type(node), pointer :: new_node
    type(node), pointer :: current

    allocate(new_node)
    new_node%id = id
    new_node%nombre = nombre

    ! Llamamos al procedimiento push de la pila para insertar el tipo de imagen
    call new_node%stack%push_i(tipo_img, id)

    if (.not. associated(self%head)) then
        self%head => new_node
    else
        current => self%head
        do while (associated(current%next))
            current => current%next
        end do
        current%next => new_node
    end if
end subroutine push_imp
```

Detalles adicionales:

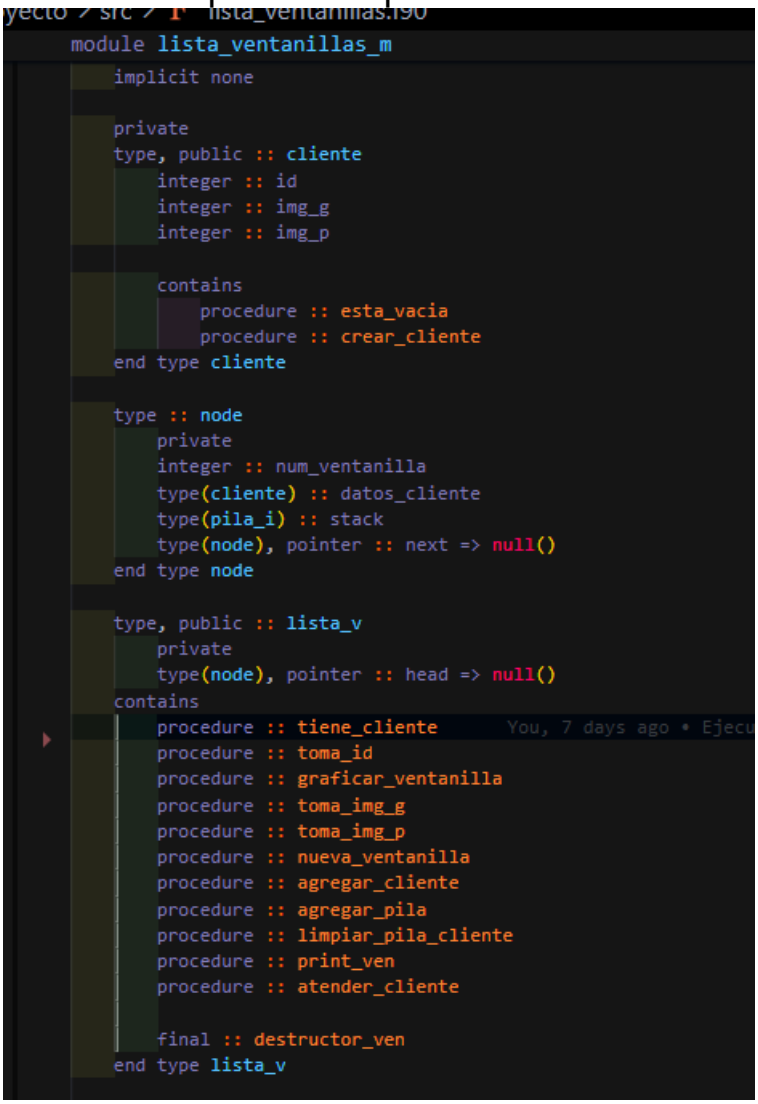
La subrutina push_imp agrega un nuevo nodo a la cola de impresoras y también inserta un tipo de imagen en la pila asociada al nodo utilizando la subrutina push_i definida en el módulo pila_img_m.
La subrutina print_imp llama a la subrutina print_i de la pila asociada a cada nodo para mostrar los tipos de imagen almacenados en la pila.

```
    subroutine pop_imp(self)
        class(cola_imp), intent(inout) :: self
        type(node), pointer :: temp

        if (.not. associated(self%head)) then
            print *, "La cola está vacía. No se puede realizar 'pop'."
            return
        else
            temp => self%head
            self%head => self%head%next
            deallocate(temp)
        end if
    end subroutine pop_imp

    subroutine print_imp(self)
        class(cola_imp), intent(in) :: self
        type(node), pointer :: current
        current => self%head

        print *, "Elementos en la cola de impresoras:"
        do while (associated(current))
            print *, "ID:", current%id, "Nombre:", current%nombre
            ! Llamamos al procedimiento print de la pila para mostrar el tipo de imagen
            call current%stack%print_i()
            current => current%next
        end do
    end subroutine print_imp

    subroutine destructor_imp(self)
        type(cola_imp), intent(inout) :: self
        type(node), pointer :: aux

        do while(associated(self%head))
            aux => self%head%next
            deallocate(self%head)
            self%head => aux
        end do
    end subroutine destructor_imp
```

Lista_ventanillas

Declaraciones y tipos:

Se define el tipo cliente que representa la información de un cliente, incluyendo su id, img_g (cantidad de imágenes grandes) y img_p (cantidad de imágenes pequeñas).
Se define el tipo node que representa un nodo en la lista de ventanillas. Cada nodo contiene un número de ventanilla, los datos del cliente asociado y una pila de imágenes representada por el tipo pila_i.
Se declara el tipo lista_v que representa la lista de ventanillas. Esta contiene un puntero al primer nodo de la lista (head).



Subrutinas:

nueva_ventanilla: Agrega una nueva ventanilla a la lista con un

número de ventanilla especificado.
agregar_cliente: Agrega un cliente a una ventanilla específica.
limpiar_pila_cliente: Vacía la pila de imágenes y los datos del cliente asociados a cada ventanilla.
atender_cliente: Atiende a los clientes procesando sus imágenes y colocándolas en la pila correspondiente de la ventanilla.
print_ven: Imprime información sobre todas las ventanillas, incluyendo el número de ventanilla, el cliente actual y las imágenes en la pila.
graficar_ventanilla: Genera un gráfico de la lista de ventanillas utilizando el formato DOT y lo convierte en una imagen PNG.
Otras subrutinas como toma_id, toma_img_g, toma_img_p, crear_cliente, esta_vacia, tiene_cliente que realizan tareas específicas relacionadas con la gestión de clientes y ventanillas.
La subrutina atender_cliente es responsable de procesar las imágenes de los clientes y colocarlas en la pila correspondiente de la ventanilla.
La subrutina limpiar_pila_cliente vacía la pila de imágenes y los datos del cliente asociados a cada ventanilla después de que se ha atendido al cliente.
La subrutina graficar_ventanilla genera un gráfico que muestra la relación entre las ventanillas, los clientes y las pilas de imágenes asociadas.

```
contains
function itoa(number) result(str)
    integer, intent(in) :: number
    character(len=14) :: str

    ! Convierte el entero a una cadena de caracteres
    write(str, '(I14)') number

end function itoa
```

```
subroutine graficar_ventanilla(self, nombre_archivo)
    implicit none
    class(lista_v), intent(in) :: self
    character(len=*), intent(in) :: nombre_archivo
    integer :: io

    integer :: index
    character(len=100), allocatable :: command
    character(:), allocatable :: connections
    character(:), allocatable :: firsts
    character(len=8) :: name
    type(node), pointer :: current

    current => self%head
    command = "dot -Tpng " // trim(nombre_archivo) // " -o " // trim(nombre_archivo) // ".png"
    io = 1
    index = 0

    connections = ""
    firsts = ""

    open(newunit=io, file=trim(nombre_archivo))
    write(io, *) "digraph ListaVentanillas {"
    write(io, *) "  rankdir=LR;"

    if (associated(self%head)) then
        do while (associated(current))
            write(name, '(I5)') current%num_ventanilla

            if (firsts == "") then
                firsts = trim(name)
            end if

                ', Img_p: ' // trim(itoa(current%datos_cliente%img_p)) // &
                '", shape=box];'

            ! Generar etiqueta para el nodo de la ventana
            write(io, '(A)') '"nodo' // trim(name) // '"[label="Ventanilla: ' // trim(name) // '", shape=box];'

            ! Generar conexión entre el cliente y la ventana
            connections = connections // '"cliente' // trim(name) // '" -> ' // '"nodo' // trim(name) // '";'

! Generar etiqueta para el nodo de la pila de ventanilla
! Obtener los elementos de la pila de la ventanilla actual
            write(io, '(A)') '"pila' // trim(name) // '"[label="Pila de Ventanilla ' // trim(name) // &
            '& : ' // trim(current%stack%get_elements()) // '", shape=box];'

        ! Generar conexión entre la ventana y la pila
        connections = connections // '"nodo' // trim(name) // '" -> ' // '"pila' // trim(name) // '";'


Code Suggestions

            current => current%next
            index = index + 1
        end do
    end if

    write(io, *) connections
    write(io, *) "}"
    close(io)

    call execute_command_line(command, exitstat=io)

    if (io /= 0) then
        print *, "Ocurrió un error al generar la imagen."
    else
        print *, "Imagen generada satisfactoriamente."
    end if
end subroutine graficar_ventanilla
```

```
integer function toma_id(self, ventanilla)
    class(lista_v), intent(in) :: self
    integer, intent(in) :: ventanilla
    type(node), pointer :: current
    current => self%head
    do while(associated(current))
        if (current%num_ventanilla == ventanilla) then
            toma_id = current%datos_cliente%id
            return
        end if
        current => current%next
    end do
end function toma_id

integer function toma_img_g(self, ventanilla)
    class(lista_v), intent(in) :: self
    integer, intent(in) :: ventanilla
    type(node), pointer :: current
    current => self%head
    do while(associated(current))
        if (current%num_ventanilla == ventanilla) then
            toma_img_g = current%datos_cliente%img_g
            return
        end if
        current => current%next
    end do
end function toma_img_g
integer function toma_img_p(self, ventanilla)
    class(lista_v), intent(in) :: self
    integer, intent(in) :: ventanilla
    type(node), pointer :: current
    current => self%head
    do while(associated(current))
        if (current%num_ventanilla == ventanilla) then
            toma_img_p = current%datos_cliente%img_p
            return
        end if
        current => current%next
    end do
end function toma_img_p
```

```
    end function toma_img_p
    subroutine crear_cliente(self, id, img_g, img_p)
        class(cliente), intent(inout) :: self
        integer, intent(in) :: id
        integer, intent(in) :: img_g
        integer, intent(in) :: img_p
        self%id = id
        self%img_g = img_g
        self%img_p = img_p
    end subroutine crear_cliente

    logical function esta_vacia(self)
    class(cliente), intent(in) :: self
    esta_vacia = (self%id == 0)
end function esta_vacia


    ! Método para verificar si la ventanilla tiene una cliente
logical function tiene_cliente(self, ventanilla)
    class(lista_v), intent(in) :: self
    integer, intent(in) :: ventanilla
    logical :: hay_cola
    type(node), pointer :: current

    hay_cola = .false. ! Inicializa la variable a falso

    current => self%head

    do while(associated(current))
        if (current%num_ventanilla == ventanilla) then
            ! Verifica si la pila asociada a la ventanilla no está vacía
            hay_cola = .not. current%datos_cliente%esta_vacia()
            exit ! Sale del bucle una vez que se encuentra la ventanilla
        end if
        current => current%next
    end do

end function tiene_cliente
```

```
    subroutine nueva_ventanilla(self, num_ventanilla)
        implicit none
        class(lista_v), intent(inout) :: self
        integer, intent(in) :: num_ventanilla
        type(node), pointer :: aux, nuevo
        allocate(nuevo)
        nuevo%num_ventanilla = num_ventanilla
        nuevo%datos_cliente%id = 0
        nuevo%datos_cliente%img_g = 0
        nuevo%datos_cliente%img_p = 0
        nuevo%next => null()
        if(associated(self%head)) then
            aux => self%head
            do while(associated(aux%next))
                aux => aux%next
            end do
            aux%next => nuevo
        else
            self%head => nuevo
        end if
    end subroutine nueva_ventanilla


  ! Método para limpiar la pila y la cliente de la ventanilla
    subroutine limpiar_pila_cliente(self)
        implicit none
        class(lista_v), intent(inout) :: self
        type(node), pointer :: aux
        if(associated(self%head)) then
            aux => self%head
            do while(associated(aux))
                call aux%stack%vaciar_i()   ! Vaciar la pila utilizando el nuevo método
                print*, "Ventanilla:", aux%num_ventanilla
                aux => aux%next
            end do
        end if
    end subroutine limpiar_pila_cliente
```

Fortran Program Manager

FPM (Fortran Package Manager) es una herramienta que se utiliza para gestionar paquetes en el lenguaje de programación Fortran. Similar a otros administradores de paquetes como pip para Python, npm para Node.js, o cargo para Rust, FPM simplifica la instalación, actualización y gestión de bibliotecas y programas escritos en Fortran.

Aquí hay una descripción de las principales características y funciones de FPM:

Gestión de Dependencias: FPM maneja las dependencias de los paquetes Fortran de manera eficiente, permitiendo que los usuarios instalen fácilmente las bibliotecas que necesitan para sus proyectos.

Instalación Simple: Con FPM, instalar paquetes Fortran es sencillo. Los usuarios pueden simplemente especificar el nombre del paquete y FPM se encarga de descargarlo e instalarlo automáticamente, incluyendo todas las dependencias necesarias.

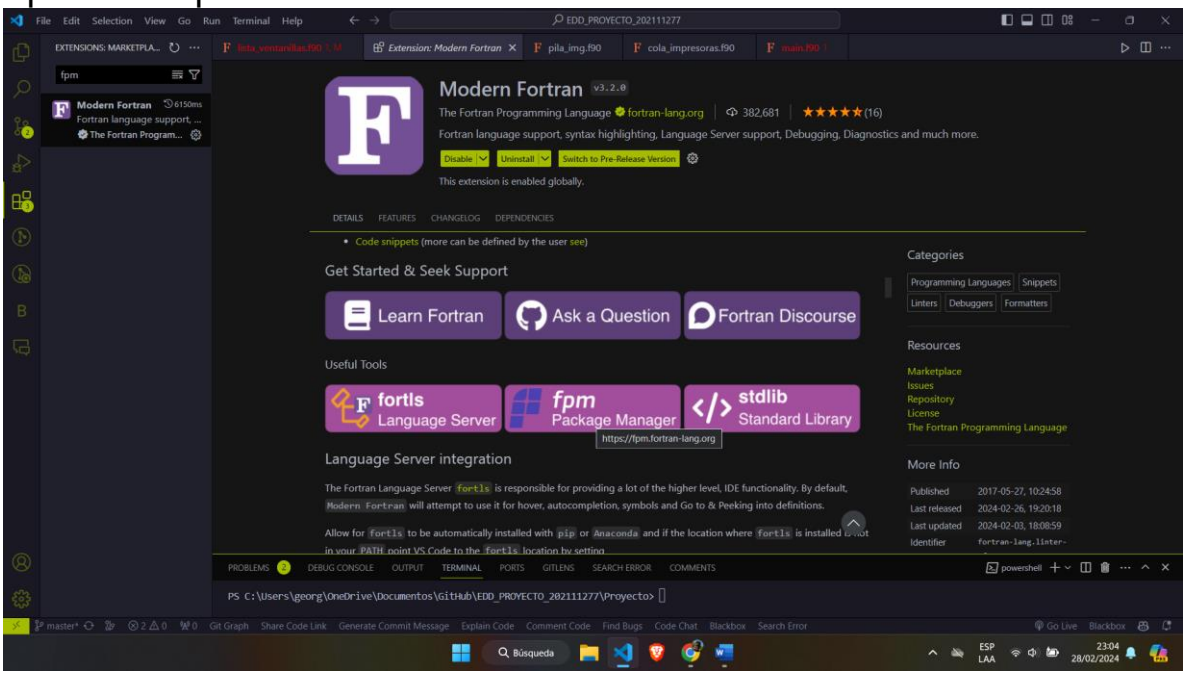
Versionamiento: FPM permite a los desarrolladores especificar las versiones de las bibliotecas que sus proyectos requieren, lo que garantiza una reproducción consistente del entorno de desarrollo.

Creación de Proyectos: FPM facilita la creación de nuevos proyectos Fortran al proporcionar plantillas y herramientas para la organización de la estructura del proyecto.

Integración con Repositorios: FPM se integra con repositorios de paquetes Fortran populares, como GitHub, GitLab y Bitbucket, lo que permite a los usuarios buscar y utilizar una amplia gama de bibliotecas disponibles en línea.

Gestión de Dependencias del Sistema: Además de gestionar las dependencias específicas del proyecto, FPM también puede interactuar con el sistema operativo para instalar bibliotecas y dependencias del sistema necesarias para compilar y ejecutar programas Fortran.

Fpm forma parte de la extensión Modern Fortran



Mas información en: https://fpm.fortran-lang.org/

Graphviz

Graphviz es una potente herramienta de visualización de gráficos que permite representar de manera visual relaciones y estructuras complejas mediante grafos. Con Graphviz, los usuarios pueden crear diagramas y representaciones visuales de datos que son difíciles de entender o visualizar simplemente con texto o tablas.

Representación de Grafos: Graphviz se utiliza principalmente para representar grafos, que son estructuras de datos compuestas por nodos y aristas que conectan los nodos. Estos grafos pueden ser dirigidos o no dirigidos, ponderados o no ponderados, y representan una amplia gama de relaciones y estructuras.

Lenguaje de Descripción de Gráficos (DOT): Graphviz utiliza un lenguaje de descripción de gráficos llamado DOT. DOT es un lenguaje simple y legible que permite a los usuarios describir la estructura y las propiedades de los grafos de una manera clara y concisa.