Estudio
Shonos

SISTEMA DE
GESTION
De Procesos

```cpp
#include <iostream>
#include <string>
#include <fstream>
#include <windows.h> // Para usar funciones de Windows como Sleep()
#define MAX 5

using namespace std;
```

```cpp
struct Proceso { // Estructura que representa un proceso
    // Atributos del proceso
    int id;
    string nombre;
    int prioridad;
    Proceso* siguiente; // Puntero al siguiente proceso en la lista

    // Constructor para inicializar un nuevo proceso
    Proceso(int ID, string Nombre, int Prioridad) {
        id = ID;
        nombre = Nombre;
        prioridad = Prioridad;
        siguiente = NULL; // Inicializa el puntero siguiente como NULL
    }
};
```

```cpp
// Pila de memoria para los procesos

Proceso* pila[MAX]; // Arreglo que representa la pila de memoria
int tope = -1; // Variable que indica el índice del último elemento en la pila

// Cola para gestionar la memoria

Proceso* cola[MAX]; // Arreglo que representa la cola de memoria
int frente = -1; // Índice del primer elemento en la cola
int final = -1; // Índice del último elemento en la cola
```

```
case 1:
    // Pedimos al usuario cuantos procesos desea registrar
    cout << "Cuantos procesos desea registrar? "; cin >> NumEl;

    // Ingresamos los datos de cada proceso
    for (int i = 0; i < NumEl; i++) {
    cout << "\nRegistro del proceso " << (i + 1) << endl;
    // Pedimos al usuario que ingrese los datos del proceso
    do {
        cout << "Ingrese el ID del proceso: "; cin >> id;

        if (id <= 0) { // Verifica que el ID sea un número positivo
            cout << "El ID debe ser un número positivo. Intente de nuevo." << endl;
        }
    } while (id <= 0);

    cout << "Ingrese el nombre del proceso: "; cin >> nombre;

    do {
        cout << "Ingrese la prioridad del proceso (1-10): "; cin >> prioridad;

        if (prioridad < 1 || prioridad > 10) { // Verifica que la prioridad esté en el rango válido
            cout << "La prioridad debe estar entre 1 y 10. Intente de nuevo." << endl;
        }
    } while (prioridad < 1 || prioridad > 10);

    // Llamamos a la función para agregar el proceso a la lista
    agregarAlFinal(listaProcesos, id, nombre, prioridad);
    }
    break;
```

Este código se encarga de:
Preguntar cuántos procesos quiere registrar el usuario.
Para cada proceso:
Pedir un ID (debe ser positivo).
Pedir un nombre.
Pedir una prioridad (entre 1 y 10).
Agregar ese proceso a una lista usando una función llamada agregarAlFinal.

```
void agregarAlFinal(Proceso*& listaProcesos, int ID, string Nombre, int Prioridad) {
    Proceso *nuevoProceso = new Proceso(ID, Nombre, Prioridad); // Crea un nuevo proceso con los datos proporcionados

    if (listaProcesos == NULL) { // Verifica si la lista está vacía
        listaProcesos = nuevoProceso; // Si está vacía, el nuevo proceso se convierte en el primer elemento de la lista
    } else {
        Proceso* temp = listaProcesos; // Crea un puntero temporal para recorrer la lista
        while (temp->siguiente) { // Recorre la lista hasta el final
            temp = temp->siguiente;
        }
        temp->siguiente = nuevoProceso; // Asigna el nuevo proceso al final de la lista
        cout << "Proceso agregado al final de la lista.\n";
    }
}
```

Este código se encarga de registrar múltiples procesos en una lista enlazada. Primero, le pregunta al usuario cuántos procesos desea registrar. Luego, para cada proceso, solicita un ID (que debe ser un número positivo), un nombre y una prioridad (que debe estar entre 1 y 10). Una vez validados los datos, cada proceso se agrega al final de la lista mediante la función agregarAlFinal.

```cpp
void imprimirProcesos(Proceso* listaProcesos) {
    Proceso* temp = listaProcesos; // Crea un puntero temporal para recorrer la lista

    if (!temp) { // Verifica si la lista está vacía
        cout << "No hay procesos registrados.\n"; // Mensaje si la lista está vacía
        return;
    }
    cout << "Lista de procesos:\n";
    while (temp != NULL) { // Recorre la lista e imprime los datos de cada proceso
        cout << "ID: " << temp->id << ", Nombre: " << temp->nombre << ", Prioridad: " << temp->prioridad << endl;
        temp = temp->siguiente; // Avanza al siguiente proceso
    }
    cout << endl;
}
```

La función imprimirProcesos muestra en pantalla todos los procesos registrados en una lista enlazada. Comienza creando un puntero temporal que recorre la lista desde el inicio. Si la lista está vacía, muestra un mensaje indicando que no hay procesos registrados y termina la función. En caso contrario, recorre cada nodo de la lista y muestra el ID, nombre y prioridad de cada proceso, uno por uno, hasta llegar al final de la lista.

```
271  void buscarProceso(Proceso*& listaProcesos, int id) {
272      Proceso* temp = listaProcesos;
273      // Indicamos que comienza desde el inicio de la lista
274      while (temp) {
275          if (temp->id == id) {
276              cout << "ID: " << temp->id << ", Nombre: " << temp->nombre << ", Prioridad: " << temp->prioridad << endl;
277              return;
278          }
279          temp = temp->siguiente;
280      }
281      // Cuando encuentre el ID que busquemos lo imprimirá
282      cout << "Proceso no encontrado.\n";
283      // Nos informará si no lo encuentra
284  }
```

La función buscarProceso permite encontrar e imprimir la información de un proceso específico dentro de una lista enlazada, buscando por su ID. Recorre la lista desde el inicio utilizando un puntero temporal, comparando el ID de cada nodo con el valor buscado. Si encuentra un proceso con el ID coincidente, muestra en pantalla su ID, nombre y prioridad, y finaliza la búsqueda. Si recorre toda la lista sin encontrar coincidencias, muestra un mensaje indicando que el proceso no fue encontrado.

```
250  void eliminarProceso(Proceso*& listaProcesos, int id) {
251      Proceso* temp = listaProcesos;
252      // Creamos un puntero que buscara el proceso que indiquemos
253      Proceso* anterior = NULL;
254      // Creamos otro puntero que guardará el nodo anterior que deseqamos eliminar
255      while (temp && temp->id != id) {
256          anterior = temp;
257          temp = temp->siguiente;
258      }
259      // Con esto podemos eliminar un proceso de la lista cuando el ID coincide
260      if (!temp) {
261          cout << "Proceso no encontrado.\n";
262          return;
263      }
264      // Esto informa cuando no se encuentre ningun proceso con ese ID y termina la función
265      if (!anterior) listaProcesos = temp->siguiente;
266      else anterior->siguiente = temp->siguiente;
267      delete temp;
268      // Esto hace que libere la memoria del proceso eliminado
269      cout << "Proceso eliminado.\n";
270  }
```

La función eliminarProceso permite eliminar un nodo (proceso) de una lista enlazada buscando por su ID. Recorre la lista con un puntero temp para ubicar el proceso, y utiliza otro puntero anterior para mantener la referencia al nodo previo. Si no se encuentra el proceso, se muestra un mensaje indicándolo y la función termina. Si el proceso a eliminar es el primero de la lista, se actualiza el puntero principal para que apunte al siguiente nodo. Si es un nodo intermedio o final, se ajustan los enlaces para excluirlo. Finalmente, se libera la memoria del proceso eliminado y se informa al usuario.

```
286  void modificarProceso(Proceso*& listaProcesos, int id) {
287      Proceso* temp = listaProcesos;
288      // Creamos un puntero que buscara el proceso que indiquemos
289      while (temp) { // Recorremos la lista de procesos
290          if (temp->id == id) { // Verificamos si el ID del proceso coincide con el ID buscado
291              // Si encontramos el proceso, solicitamos los nuevos datos
292              cout << "Ingrese el nuevo nombre del proceso: ";
293              cin >> temp->nombre;
294              cout << "Ingrese la nueva prioridad del proceso: ";
295              cin >> temp->prioridad;
296              cout << "Proceso modificado exitosamente.\n";
297              return;
298          }
299          temp = temp->siguiente; // Avanzamos al siguiente proceso en la listaS
300      }
301      // Cuando encuentre el ID que busquemos lo modificará
302      cout << "Proceso no encontrado.\n";
303      // Nos informará si no lo encuentra
304  }
```

La función modificarProceso permite actualizar los datos de un proceso dentro de una lista enlazada, buscándolo por su ID. Utiliza un puntero temporal para recorrer la lista y, si encuentra un nodo con el ID especificado, solicita al usuario que ingrese un nuevo nombre y una nueva prioridad, actualizando esos valores directamente en el nodo encontrado. Si no se encuentra un proceso con ese ID, se muestra un mensaje indicando que el proceso no fue hallado.

```
134  void insertar(Proceso*& listaProcesos){
135      if (tope == MAX - 1) { // Verifica si la pila está llena
136          cout << "Memoria llena. No se puede insertar mas elementos." << endl;
137      } else {
138          int id2;
139          cout << "Ingrese el ID del proceso a asignar a la memoria: "; cin >> id2;
140          Proceso* temp = listaProcesos; // Crea un puntero temporal para recorrer la lista de procesos
141          while (temp != NULL) { // Recorre la lista de procesos
142              if (temp->id == id2) { // Verifica si el ID del proceso coincide con el ID ingresado
143                  pila[++tope] = temp; // Agrega el proceso a la pila
144                  cout << "Proceso con ID " << id2 << " agregado a la memoria." << endl;
145                  encolar(); // Llama a la función para encolar el proceso en la cola de memoria
146                  return; // Sale de la función una vez que se ha agregado el proceso
147              }
148              temp = temp->siguiente; // Avanza al siguiente proceso en la lista
149          }
150          cout << "No se encontró un proceso con ese ID." << endl;
151      }
152  }
```

La función insertar, permite asignar un proceso específico desde la lista general de procesos a la memoria, primero omprueba si la pila de memoria está llena, luego pide al usuario que ingrese el ID del proceso que desea mover a la memoria. Hace uso de un puntero temporal para buscar el ID en la lista de procesos, si el ID es valido el proceso se almacena en la pila, por ultimo llama a la función encolar para su ejecución.

```
156  void eliminarDePila() {
157      if (tope == -1) { // Verifica si la pila está vacía
158          cout << "La pila de memoria está vacía. No se puede eliminar ningún proceso." << endl;
159          return;
160      }
161      cout << "Procesos en la pila de memoria:" << endl;
162      for (int i = 0; i <= tope; i++) { // Recorre la pila e imprime los IDs de los procesos
163          cout << "ID: " << pila[i]->id << ", Nombre: " << pila[i]->nombre << ", Prioridad: " << pila[i]->prioridad << endl;
164      }
165      int idEliminar;
166      cout << "Ingrese el ID del proceso que desea quitar de la pila: ";
167      cin >> idEliminar;
168
169      int pos = -1; // Variable para almacenar la posición del proceso a eliminar
170      for (int i = 0; i <= tope; i++) { // Recorre la pila desde el inicio hasta el tope
171          if (pila[i]->id == idEliminar) { // Busca el proceso con el ID especificado
172              pos = i; // Guarda la posición del proceso a eliminar
173              break;
174          }
175      }
176      if (pos == -1) {
177          cout << "No se encontró un proceso con ese ID en la pila." << endl; // Mensaje si no se encuentra el proceso
178          return;
179      }
180      Proceso* procesoEliminado = pila[pos]; // Guarda el proceso que se va a eliminar
181      // Desplazar los elementos para llenar el hueco
182      for (int i = pos; i < tope; i++) { // Desplaza los elementos hacia la izquierda
183          pila[i] = pila[i + 1]; // Mueve los elementos hacia la izquierda
184      }
185      tope--; // Decrementa el índice del tope de la pila
186      cout << "Proceso con ID " << procesoEliminado->id << " eliminado de la memoria." << endl;
187      eliminarDeColaPorID(idEliminar); // <-- sincroniza la cola
188  }
```

La función eliminarDePila permite remover un proceso específico de la pila de memoria, primero comprueba si la pila está vacía, luego muestra todos los procesos actualmente en la pila, solicita al usuario el ID de la pila que desea eliminar y busca una coincidencia con el valor introducido y si llega a encontrarlo guarda la posición del proceso, si no encuentra el proceso informa al usuario.

Guarda una referencia temporal del proceso a eliminar, después reorganiza la pila desplazando los elementos para llenar el espacio dejado por el proceso eliminado y por ultimo reduce el contador de elementos en la pila.

```
192  void liberarMemoria() {
193      if (tope == -1) { // Verifica si la pila está vacía
194          cout << "La pila de memoria está vacía. No hay procesos para liberar." << endl;
195          return;
196      }
197      // Solo vacía la pila, no elimina los procesos de la lista enlazada
198      tope = -1; // Reinicia el tope de la pila a -1 para indicar que está vacía
199      cout << "Todos los procesos han sido quitados de la memoria (pila) exitosamente." << endl;
200      vaciarCola(); // <-- sincroniza la cola
201  }
```

La función liberarMemoria() se encarga de vaciar la pila de memoria y sincronizar la cola de procesos. Primero verifica si la pila está vacía, mostrando un mensaje en ese caso; si contiene procesos, reinicia el índice tope a -1 y muestra un mensaje de confirmación. Además, llama a vaciarCola() para garantizar que la cola de ejecución también quede vacía y sincronizada con el estado de la pila. En esencia, esta función actúa como un reinicio rápido, dejando ambos estructuras listas para nuevos procesos, pero sin eliminar los datos de la lista enlazada subyacente.

Por ejemplo, si la pila tenía 3 procesos, después de ejecutarla, tanto la pila como la cola quedarán vacías y listas para reutilizarse.

```
205  void visualizarPila() {
206      if (tope == -1) { // Verifica si la pila está vacía
207          cout << "La pila de memoria está vacía." << endl; // Mensaje si la pila está vacía
208      } else {
209          cout << "Procesos en la pila de memoria:" << endl;
210          for (int i = 0; i <= tope; i++) { // Recorre la pila e imprime los IDs de los procesos
211              cout << "ID: " << pila[i]->id << ", Nombre: " << pila[i]->nombre << ", Prioridad: " << pila[i]->prioridad << endl;
212          }
213      }
214  }
```

Este código muestra los procesos almacenados en una pila de memoria. Primero verifica si la pila está vacía, mostrando un mensaje en ese caso. Si hay procesos, los recorre desde la base hasta el tope e imprime los detalles de cada uno: ID, nombre y prioridad. Esencialmente, funciona como una lista donde se ven todos los procesos apilados, desde el más antiguo hasta el más reciente.

Por ejemplo, si hay tres procesos, imprime sus datos en orden de inserción.

Simple y directo: muestra lo que hay en la pila o avisa si no hay nada.

```
43  void encolar() {
44      if (tope == -1) { // Verifica si la pila está vacía
45          cout << "La pila de memoria está vacía. No se puede encolar." << endl;
46          return;
47      }
48      Proceso* nuevo = pila[tope]; // Toma el proceso del tope de la pila
49
50      if (frente == -1) { // Verifica si la cola está vacía
51          frente = final = 0; // Si la cola está vacía, inicializa el frente
52          cola[0] = nuevo; // Agrega el nuevo proceso al frente de la cola
53          return;
54      }
55
56      int pos = frente; // Comienza desde el frente de la cola
57      while (pos <= final && cola[pos]->prioridad >= nuevo->prioridad) { // Compara las prioridades de los procesos
58          pos++; // Encuentra la posición correcta para insertar el nuevo proceso según su prioridad
59      }
60
61      // Desplazar los elementos para hacer espacio
62      for (int i = final + 1; i > pos; i--) { // Organiza la cola
63          cola[i] = cola[i - 1]; // Mueve los elementos hacia la derecha
64      }
65      cola[pos] = nuevo; // Inserta el nuevo proceso en la posición encontrada
66      final++; // Incrementa el índice del final de la cola
67  }
```

El código encolar() toma un proceso de la pila de memoria y lo inserta en la cola de CPU según su prioridad. Primero verifica si la pila está vacía; si lo está, muestra un error. Luego, si la cola está vacía, coloca el proceso al frente. Si no, busca la posición correcta comparando prioridades (los de mayor prioridad van primero), desplaza los procesos existentes para hacer espacio, inserta el nuevo proceso y actualiza el índice final.

Por ejemplo:

* Si la cola tiene [ProcesoA (3), ProcesoB (1)] y se agrega ProcesoC (2), el resultado sería [ProcesoA (3), ProcesoC (2), ProcesoB (1)].

```
71  void eliminarDeColaPorID(int idEliminar) {
72      if (frente == -1) { // Verifica si la cola está vacía
73          return;
74      }
75      int pos = -1; // Variable para almacenar la posición del proceso a eliminar
76      for (int i = frente; i <= final; i++) { // Recorre la cola desde el frente hasta el final
77          if (cola[i]->id == idEliminar) { // Busca el proceso con el ID especificado
78              pos = i; // Guarda la posición del proceso a eliminar
79              break;
80          }
81      }
82      if (pos == -1) return; // Si no se encontró el proceso, sale de la función
83      for (int i = pos; i < final; i++) { // Desplaza los elementos hacia la izquierda para llenar el hueco
84          cola[i] = cola[i + 1]; // Mueve los elementos hacia la izquierda
85      }
86      final--; // Decrementa el índice del final de la cola
87      if (final < frente) frente = final = -1; // Si la cola queda vacía, reinicia los índices
88  }
```

Este código elimina un proceso específico de la cola según su ID. Primero verifica si la cola está vacía, en caso termina sin hacer nada. Si hay procesos, busca el que coincida con el idEliminar recorriendo la cola desde frente hasta final. Si lo encuentra, guarda su posición en pos; si no, sale de la función. Luego, desplaza todos los elementos posteriores una posición hacia la izquierda para rellenar el espacio dejado por el proceso eliminado, actualiza el índice final reduciéndolo en 1 y, si la cola queda vacía, reinicia ambos índices a -1. En esencia, es como quitar una persona de una fila y cerrar el hueco para que no queden espacios vacíos.

```
92      void vaciarCola() {
93          frente = final = -1; // Reinicia los índices de la cola para indicar que está vacía
94      }
```

Este código vaciarCola reinicia la cola de procesos asignando -1 a los índices frente y final, lo que indica que la cola está vacía. Al establecer estos valores, se elimina simbólicamente todo su contenido, ya que -1 es un marcador común para señalar que no hay elementos. Esto permite que la cola pueda reutilizarse desde cero, como si fuera nueva, y cuando se agregue un proceso, los índices se actualizarán correctamente. En resumen, es como borrar una lista de tareas pendientes para comenzar de nuevo.

```
120  void visualizarCola() {
121      if (frente == -1) { // Verifica si la cola está vacía
122          cout << "La cola de CPU está vacía." << endl;
123          return;
124      } else {
125          cout << "Orden de ejecucion" << endl;
126          for (int i = frente; i <= final; i++) { // Recorre la cola desde el frente hasta el final
127              cout << "ID: " << cola[i]->id << ", Nombre: " << cola[i]->nombre << ", Prioridad: " << cola[i]->prioridad << endl;
128          }
129      }
130  }
```

Este código muestra cómo se visualizan los procesos en una cola de CPU. Primero, verifica si la cola está vacía (cuando frente == -1). Si está vacía, imprime un mensaje diciendo "La cola de CPU está vacía". Si hay procesos, recorre la cola desde el frente , hasta el final y muestra los datos de cada proceso: su ID, nombre y prioridad. Básicamente, es como imprimir una lista de tareas pendientes en orden.

```
98  void ejecutarProcesosEnCola() {
99      if (frente == -1) { // Verifica si la cola está vacía
100         cout << "No hay procesos en la cola para ejecutar." << endl;
101         return;
102     }
103     cout << "Ejecutando procesos en orden de prioridad:" << endl;
104     while (frente != -1) { // Mientras haya procesos en la cola
105         cout << "Ejecutando -> ID: " << cola[frente]->id // Imprime el ID del proceso que se está ejecutando
106              << ", Nombre: " << cola[frente]->nombre // Imprime el nombre del proceso
107              << ", Prioridad: " << cola[frente]->prioridad << endl; // Imprime la prioridad del proceso
108         Sleep(1000); // Pausa de 1 segundo
109         if (frente == final) { // Verifica si es el último proceso en la cola
110             frente = final = -1; // Cola vacía
111         } else {
112             frente++; // Avanza al siguiente proceso en la cola
113         }
114     }
115     cout << "Todos los procesos han sido ejecutados." << endl;
116 }
```

El código muestra cómo se ejecutan procesos guardados en una cola. Primero, revisa si hay procesos pendientes. Si no hay, avisa que no hay nada que hacer. Si hay, va sacando uno por uno, mostrando su información, espera un segundo y pasa al siguiente. Cuando ya no quedan más, avisa que terminó. Básicamente, es como atender turnos en orden.