PL I

Tatiana Garcia Vergara

1. Descripción de los TAD´s implementados y estructuras de datos definidas.

Clase Pila: tiene un puntero a un nodo de forma privada, que representa la parte superior de la pila, que es donde se realizan las inserciones, las consultas y las eliminaciones. Hay metodos para construir y destruir pilas, ademas de otras operaciones comunes de las pilas como apilar, desapilar y obtener el tamaño de la pila. Tambien, se ha creado un metodo para imprimir la pila.

Clase Cola: tiene dos diferente punteros de forma privada: uno al primero, que es donde se van a realizar las eliminaciones y consultas, y otro al final, que es donde se van a realizar las inserciones. Hay metodos para construir y destruir colas, al igual que otras operaciones basicas como encolar, desencolar y obtener el primero de la cola. Al igual que en las pilas, se ha creado un metodo para imprimir las colas.

```
45  class Cola
46  {
47      public:
48          Cola() : frente(NULL), final(NULL) {}
49          ~Cola();
50          void encolar(Caja v);
51          Caja desencolar();
52          void mostrarCola();
53          Caja primero();
54      private:
55          pNodo frente, final;
56  };
57
58  class Pila
59  {
60      private:
61          pNodo cima;
62      public:
63          Pila() : cima(NULL) {}          //Constructo
64          ~Pila();
65      void apilar(Caja v);
66      Caja desapilar();
67      void mostrarPila();
68      int tamPila();
69  };
70
```

Clase Nodo: tiene dos atributos privados: valor, que almacena un objeto de tipo Caja y siguiente, un puntero a otro nodo de la misma clase. Además, se ha añadido la amistad con las clases Pila y Cola, para que pudiera acceder a sus atributos privados. Tambien, se ha hecho un constructor para los nodos.

```
27
28  class Nodo
29  {
30      private:
31          Caja valor;
32          Nodo *siguiente;
33          friend class Pila;
34          friend class Cola;
35      public:
36          Nodo(Caja v, Nodo *sig = NULL)
37          {
38              valor = v;
39              siguiente = sig;
40          }
41  };
42
43  typedef Nodo *pNodo;
44
```

Estructura ID:

- **destino:** una cadena de caracteres que representa el destino.
- **num:** una cadena de caracteres que, en este caso, van a ser números.
- **origen:** una cadena de caracteres que representa el origen.

Estructura Fecha:

- **mes:** número entero que representa un mes.
- **anno:** número entero que representa un año.

Estructura Caja:

- **id:** una instancia de la estructura ID.
- **contenido**: una cadena de caracteres que representa el contenido de la caja.
- **centro_ref:** una cadena de caracteres que representa el centro de referencia.
- **fechaCad:** una instancia de la estructura Fecha, que representa la fecha de caducidad del contenido de la caja.

```
 6
 7  struct ID
 8  {
 9      string destino;
10      string num;
11      string origen;
12  };
13
14  struct Fecha
15  {
16      int mes;
17      int anno;
18  };
19
20  struct Caja
21  {
22      ID id;
23      string contenido;
24      string centro_ref;
25      Fecha fechaCad;
26  };
27
```

2. Explicación del funcionamiento del programa y de los métodos/funciones implementadas.

El programa crea ocho cajas o ninguna caja de forma aleatoria y las guarda en una pila que simula ser una furgoneta, y hace esto cuatro veces con cuatro diferentes furgonetas. Luego, las cajas de las furgonetas se meten en una cola, que simula ser un centro de distribución y finalmente, las cajas se reparten en tres pilas, que simulan ser camiones, según el destino que tengan las cajas. Cuando los camiones están llenos, se vacían.

El programa repite esto en un bucle, en este caso, hasta diez veces.

Las funciones que han sido implementadas son:

- **Funciones para la creación de las cajas**: las cajas se crean de forma aleatoria, así que hay métodos para crear cada elemento de una caja de forma aleatoria y luego otro método, que junta todos los anteriores para hacer la caja.

    - **calcularPosicionAleatoria(int n)** -> calcula un número aleatorio, siendo el máximo el número entero que recibe como argumento.

    - **fechaAleatoria()** -> calcula un mes y un año de forma aleatoria, y crea un fecha. Los años que se crean están en un rango de 2023 a 2041, para que las fechas de caducidad sean un poco más realistas.

    - **idAleatorio(string origen)** -> crea cuatro números aleatorios y los junta como una cadena de caracteres. También, elige un destino de forma aleatoria de una lista de posibles destinos y crea un ID. El origen lo coge de la cadena de caracteres que recibe como argumento.

    - **contenidoAleatorio()** -> se crea un contenido a partir de crear una posicion aleatoria y elegir el elemento que este en esa posicion en la lista de posibles contenidos.

    - **cajaAleatoria(string origen)** -> se crea una caja. Todos los elementos son creados aleatoriamente, a partir de las funciones anteriores, a excepcion del centro de referencia, que es siempre el mismo. Además, si el contenido es un objeto que no puede caducar, la fecha se convierte en 12/2100.

```
154  int calcularPosicionAleatoria(int n)
155  {
156      return rand() % n;
157  }
158
159  Fecha fechaAleatoria()
160  {
161      Fecha fecha;
162      fecha.mes = calcularPosicionAleatoria(12) + 1;
163      fecha.anno = calcularPosicionAleatoria(20) + 2023;
164
165      return fecha;
166  }
167
168  ID idAleatorio(string origen)
169  {
170      string numeros = "";
171
172      for (int i = 0; i < 4; i++)
173      {
174          numeros = numeros + to_string(rand() % 10);
175      }
176
177      string destinos[] = {"MAR", "LIS", "GRE"};
178      string destino = destinos[rand() % 3];
179
180
181      ID id = {destino, numeros, origen};
182      return id;
183  }
```

```
184
185     string contenidoAleatorio()
186     {
187         string contenidosPosibles[] = {"harina", "pasta","legumbres", "leche", "medicinas", "higiene", "agua",
188                                        "aceite", "sal", "azucar", "galletas",
189                                       "latas_cons", "iluminacion", "herramientas",
190                                       "combustible", "tiendas_camp", "ropa", "mantas", "limpieza"};
191
192         string contenido = contenidosPosibles[calcularPosicionAleatoria(19)];
193
194         return contenido;
195
196     }
197
```

```
197
198     Caja cajaAleatoria(string origen)
199     {
200         ID id = idAleatorio(origen);
201         string contenido = contenidoAleatorio();
202         string centro = "Alcala";
203         Fecha fechaCad;
204
205         if (contenido == "iluminación" or contenido == "herramientas" or
206             contenido == "tiendas_camp" or contenido == "ropa" or contenido == "limpieza" or contenido == "mantas")
207         {
208             fechaCad = {12,2100};
209         }
210         else
211         {
212             fechaCad = fechaAleatoria();
213         }
214
215         Caja caja;
216         caja.contenido = contenido;
217         caja.fechaCad = fechaCad;
218         caja.id = id;
219         caja.centro_ref = centro;
220
221         return caja;
222     }
```

- **Funciones para imprimir cabeceras**: hay una función para las furgonetas, para el centro de distribución y para los camiones. Para poder usarlas para las cuatro diferentes furgonetas, el origen de la furgoneta se recibe como argumento en la función, al igual que los camiones reciben el destino al que van las cajas.

```
void imprimirCabeceraFurgoneta(string origen)
{
    cout << "|" << "-----------------------------------------------------------------" << "|" << endl;
    cout << "|" << setw(40) << "FURGONETA DE " << origen << setw(25) << "|" << endl;
    cout << "|" << "-----------------------------------------------------------------" << "|" << endl;

    cout << "|" << setw(7) << "ID" << setw(8) << "|" << setw(12) << "CENTRO REF." << setw(4) << "|" << setw(11) << "CONTENIDO" << setw(4) << "|" << setw(18) << "FECHA DE CONSUMO" << setw(2)
         << "|" << endl;
    cout << "|" << "-----------------------------------------------------------------" << "|" << endl;

}

void imprimirCabeceraCentro()
{

    cout << "|" << "-----------------------------------------------------------------" << "|" << endl;
    cout << "|" << setw(46) << " CENTRO DE DISTRIBUCION " << setw(20) << "|" << endl;
    cout << "|" << "-----------------------------------------------------------------" << "|" << endl;
    cout << "|" << setw(7) << "ID" << setw(8) << "|" << setw(12) << "CENTRO REF." << setw(4) << "|" << setw(11) << "CONTENIDO" << setw(4) << "|" << setw(18) << "FECHA DE CONSUMO" << setw(2)
         << "|" << endl;
    cout << "|" << "-----------------------------------------------------------------" << "|" << endl;

}


void imprimirCabeceraCamion(string destino)
{


    cout << "|" << "-----------------------------------------------------------------" << "|" << endl;
    cout << "|" << setw(38) << " CAMION " << setw(1) << destino << setw(25) << "|" << endl;
    cout << "|" << "-----------------------------------------------------------------" << "|" << endl;

    cout << "|" << setw(7) << "ID" << setw(8) << "|" << setw(12) << "CENTRO REF." << setw(4) << "|" << setw(11) << "CONTENIDO" << setw(4) << "|" << setw(18) << "FECHA DE CONSUMO" << setw(2)
         << "|" << endl;
    cout << "|" << "-----------------------------------------------------------------" << "|" << endl;

}
```

- **Funciones para las pilas**: además de los método básicos de las pilas: desapilar y apilar, otros métodos que se han implementado son:

    - **tamPila()** -> devuelve el tamaño de la pila. Usa un puntero al nodo que esta en la cima de la pila y un contador y mientras la pila no está vacía, recorre la pila incrementando el contador en cada iteración y moviendo el puntero al siguiente nodo. Al final, devuelve el contador, que es el tamaño de la pila.

    - **mostrarPila()** -> imprime los datos de todos los objetos de la pila, en este caso, las cajas.

    -

```
int Pila::tamPila()
{
    pNodo actual = cima;
    int i = 0;
    if (!actual)
    {
        return 0;
    }
    while(actual)
    {
        i++;
        actual = actual->siguiente;
    }
    return i;
}
```

```
void Pila::mostrarPila()
{
    pNodo actual = cima;

    if (!actual)
    {
        return;
    }

    while (actual)
    {

        cout << "| " << setw(6) << actual->valor.id.destino
        << actual->valor.id.num << actual->valor.id.origen
        << setw(4) << " | " << setw(12) << actual->valor.centro_ref
        << setw(4) << " | " << setw(12) << actual->valor.contenido
        << setw(2) << " | " << setw(11) << actual->valor.fechaCad.mes
        << setw(2) << "/" << setw(4) << actual->valor.fechaCad.anno
        << " |" << endl;

        actual = actual->siguiente;
    }
}
```

- **Funciones para las colas**: además de los método básicos de las colas: encolar y desencolar, otros métodos que se han implementado son:

    - **primero()** -> devuelve el primer elemento de la cola. Se crea una caja vacia y la funcion verifica si hay algun elemento en el frente de la cola, y en caso afirmativo, devuelve el valor de la caja. Si la cola esta vacia devuelve la caja vacia.

    - **mostrarCola()** -> imprime los datos de todos los objetos de la cola, en este caso, las cajas.

```
Caja Cola::primero()
{
    Fecha fecha = {0,0};
    ID id = {"","",""};
    Caja vacio = {id,"","",fecha};

    if (frente)
    {
        return frente->valor;
    }
    else
    {
        return vacio;
    }
}
```

```
void Cola::mostrarCola()
{
    pNodo actual = frente;

    if (!actual)
    {
        return;
    }

    while (actual)
    {

        cout << "| " << setw(6) << actual->valor.id.destino <<
        actual->valor.id.num << actual->valor.id.origen << setw(4)
        << " | " << setw(12) << actual->valor.centro_ref << setw(4)
        << " | " << setw(12) << actual->valor.contenido << setw(2)
        << " | " << setw(11) << actual->valor.fechaCad.mes << setw(2)
        << "/" << setw(4) << actual->valor.fechaCad.anno << " |" << endl;

        // Avanza al siguiente nodo
        actual = actual->siguiente;
    }
}
```

3. Problemas encontrados durante el desarrollo de la práctica y solución adoptada.

No he podido visualizar el contenido de forma que todas las furgonetas se muestren en una misma línea, una al lado de otra, al igual que los camiones.

A la hora de repartir las cajas, se desaparecían cajas si el camión al que iban estaba lleno. Esto pasaba porque estaba desapilandolo y luego comprobaba si había espacio, de forma que, si no había espacio, no podía meterse al camión y acababa desapareciendo. Solucione esto, creando un método que me mostrara el primero de la cola (centro de distribución), sin desapilarlo antes, y solo desapilarlo de la cola si había espacio en el camión al que tenía que ir.