Sesión #2 de Tutorías de Estructuras de Datos

Tutor: Samir Cabrera Tabash

12 de marzo del 2025

1 Tipos de Datos Fundamentales en C++

Antes de profundizar en las estructuras de datos complejas, es importante comprender los tipos de datos y valores de retorno que utilizaremos en nuestras implementaciones:

1.1 Tipos de Datos Básicos

- **int**: Números enteros (e.g., -1, 0, 42)
- **float/double**: Números de punto flotante (e.g., 3.14159)
- **char**: Caracteres individuales (e.g., 'a', 'Z', '7')
- **bool**: Valores booleanos (true o false)
- **string**: Cadenas de texto (e.g., "Hola mundo")

1.2 Tipos de Retorno en Funciones

- **void**: La función no devuelve ningún valor
- **bool**: La función devuelve true o false (ideal para validaciones)
- **int**: La función devuelve un número entero (común para contadores)
- **string**: La función devuelve una cadena de texto
- **Tipos personalizados**: La función puede devolver objetos de clases definidas por el usuario

2 Listas Doblemente Enlazadas

2.1 Concepto y Estructura

La lista doblemente enlazada es una estructura de datos secuencial donde cada nodo contiene un valor y dos punteros: uno al nodo siguiente y otro al nodo anterior. Esta bidireccionalidad permite recorrer la lista en ambos sentidos.

```
class nodoD {
public:
    nodoD(string cancion);
    nodoD(string cancion, nodoD* sig, nodoD* ant);
private:
    string cancion;
    nodoD* siguiente;
    nodoD* anterior;
    friend class ReproductorMP3;
};
```

Listing 1: Definición de la clase nodoD

2.2 Características Principales

- **Navegación bidireccional**: Permite moverse hacia adelante y hacia atrás en la lista
- **Referencias a primero y último**: Generalmente se mantienen punteros al primer y último elemento para operaciones eficientes
- **Mayor consumo de memoria**: Cada nodo almacena dos punteros en lugar de uno
- **Operaciones más eficientes**: Facilita eliminaciones e inserciones en cualquier posición sin necesidad de recorrer toda la lista

2.3 Implementación como Reproductor MP3

Una aplicación práctica de las listas doblemente enlazadas es un reproductor de música, donde necesitamos navegar hacia adelante y hacia atrás entre canciones.

```
class ReproductorMP3 {
public:
    ReproductorMP3();
    ~ReproductorMP3();
    void InsertarCancion(string cancion);
    bool ListaVacia();
    void MostrarPlaylist();
    void ReproducirSiguiente();
    void ReproducirAnterior();
    void ReproducirActual();
private:
    pnodoD primero;
    pnodoD ultimo;
    pnodoD actual;
};
```

Listing 2: Definición de la clase ReproductorMP3

2.4 Caso de Uso: Reproductor de Música

- El atributo `actual` mantiene referencia a la canción que se está reproduciendo
- `ReproducirSiguiente()` y `ReproducirAnterior()` aprovechan la estructura bidireccional para navegar entre canciones
- Insertar nuevas canciones es eficiente, añadiéndolas al final de la lista
- La lista representa una playlist completa, manteniendo el orden establecido

2.5 Aplicaciones en el Mundo Real

- **Navegación web**: Botones "Adelante" y "Atrás" en navegadores
- **Editores de texto**: Función de deshacer/rehacer
- **Aplicaciones multimedia**: Reproducción de música/videos con navegación bidireccional
- **Gestión de historial**: Movimiento entre estados anteriores y posteriores
- **Visualizadores de imágenes**: Navegación entre fotos de un álbum

3 Pilas (Stacks)

3.1 Concepto y Estructura

Una pila es una estructura de datos que sigue el principio LIFO (Last In, First Out - Último en entrar, primero en salir). Funciona como una pila de platos: solo se puede acceder al elemento superior.

```
class nodoP {
public:
    nodoP(string url);
    nodoP(string url, nodoP* signodo);
private:
    string url;
    nodoP* siguiente;
    friend class NavegadorHistorial;
};
```

Listing 3: Definición de la clase nodoP

3.2 Operaciones Fundamentales

- **Push**: Añade un elemento a la cima de la pila
- **Pop**: Elimina y retorna el elemento de la cima
- **Peek/Top**: Consulta el elemento de la cima sin eliminarlo
- **isEmpty**: Verifica si la pila está vacía

3.3 Implementación como Historial de Navegador

Una aplicación práctica de las pilas es el historial de navegación web, donde siempre regresamos a la página visitada más recientemente.

```
class NavegadorHistorial {
public:
    NavegadorHistorial();
    ~NavegadorHistorial();
    void VisitarURL(string url);
    bool HistorialVacio();
    void MostrarHistorial();
    void RegresarPagina();
    int TamanioHistorial();
private:
    pnodoP cima;
};
```

Listing 4: Definición de la clase NavegadorHistorial

3.4 Funcionamiento del Historial de Navegación

- `VisitarURL(string url)`: Equivale a un push, añadiendo la página visitada a la cima
- `RegresarPagina()`: Implementa un pop, eliminando la página actual y volviendo a la anterior
- `MostrarHistorial()`: Recorre la pila mostrando todas las páginas visitadas
- `TamanioHistorial()`: Cuenta cuántas páginas se han visitado

3.5 Aplicaciones en el Mundo Real

- **Navegadores web**: Botón "Atrás" en navegadores
- **Editores de texto**: Funciones de "Deshacer"
- **Evaluación de expresiones**: Cálculo de notación polaca inversa (RPN)

- **Llamadas a funciones**: El stack de llamadas del sistema operativo
- **Algoritmos de backtracking**: Resolución de laberintos o problemas como las N-reinas
- **Gestión de memoria**: Asignación de memoria en compiladores

4 Colas (Queues)

4.1 Concepto y Estructura

Una cola es una estructura de datos que sigue el principio FIFO (First In, First Out - Primero en entrar, primero en salir). Funciona como una fila de personas esperando: la primera persona en llegar es la primera en ser atendida.

```
class nodoC {
public:
    nodoC(string documento);
    nodoC(string documento, nodoC* signodo);
private:
    string documento;
    nodoC* siguiente;
    friend class ImpresoraQueue;
};
```

Listing 5: Definición de la clase nodoC

4.2 Operaciones Fundamentales

- **Enqueue**: Añade un elemento al final de la cola
- **Dequeue**: Elimina y retorna el elemento del frente de la cola
- **Front**: Consulta el elemento del frente sin eliminarlo
- **isEmpty**: Verifica si la cola está vacía

4.3 Implementación como Cola de Impresión

Una aplicación práctica de las colas es el sistema de impresión, donde los documentos se procesan en el orden en que fueron enviados.

```
class ImpresoraQueue {
public:
    ImpresoraQueue();
    ~ImpresoraQueue();
    void AgregarDocumento(string documento);
    bool ColaVacia();
    void MostrarCola();
    void ImprimirDocumento();
```

```
    int TamanioCola();
private:
    pnodoC primero;
    pnodoC ultimo;
};
```

Listing 6: Definición de la clase ImpresoraQueue

4.4 Funcionamiento de la Cola de Impresión

- `AgregarDocumento(string documento)`: Implementa un enqueue, añadiendo el documento al final de la cola
- `ImprimirDocumento()`: Realiza un dequeue, procesando el documento más antiguo
- `MostrarCola()`: Muestra todos los documentos en espera
- `TamanioCola()`: Informa cuántos documentos están en la cola de impresión

4.5 Aplicaciones en el Mundo Real

- **Sistemas de impresión**: Gestión de trabajos de impresión
- **Colas de procesos**: Planificación de tareas en sistemas operativos
- **Buffers**: Transferencia de datos entre procesos a diferentes velocidades
- **Gestión de recursos compartidos**: Acceso controlado a servicios
- **Aplicaciones de servicio al cliente**: Sistemas de tickets y turnos
- **Transmisión de datos**: Paquetes en redes de computadoras
- **Algoritmos de búsqueda BFS**: Recorrido de grafos por niveles

5 Comparación entre Estructuras de Datos

Característica	Lista Doble	Pila	Cola
Principio de acceso	Acceso dual	LIFO	FIFO
Inserción	Cualquier posición	Solo en cima	Solo al final
Eliminación	Cualquier posición	Solo de cima	Solo del frente
Navegación	Bidireccional	Unidireccional	Unidireccional
Punteros por nodo	Dos	Uno	Uno
Aplicación típica	Navegación	Historial	Procesamiento secuencial

6 Análisis de Eficiencia

La elección de la estructura de datos adecuada puede tener un impacto significativo en el rendimiento de una aplicación:

Operación	Lista Doble	Pila	Cola
Inserción al inicio	O(1)	O(1)	N/A
Inserción al final	O(1)	N/A	O(1)
Eliminación al inicio	O(1)	O(1)	O(1)
Eliminación al final	O(1)	N/A	N/A
Acceso aleatorio	O(n)	N/A	N/A
Búsqueda	O(n)	O(n)	O(n)

7 Implementación de Métodos Clave

7.1 Ejemplo: Reproductor MP3 (Lista Doblemente Enlazada)

```
void ReproductorMP3::ReproducirSiguiente() {
    if (!ListaVacia() && actual && actual->siguiente) {
        actual = actual->siguiente;
        cout << "Reproduciendo: " << actual->cancion << endl
    ;
    } else {
        cout << "No hay m s canciones despu s de la actual
    " << endl;
    }
}

void ReproductorMP3::ReproducirAnterior() {
    if (!ListaVacia() && actual && actual->anterior) {
        actual = actual->anterior;
        cout << "Reproduciendo: " << actual->cancion << endl
    ;
    } else {
        cout << "No hay canciones antes de la actual" <<
    endl;
    }
}
```

7.2 Ejemplo: Navegador (Pila)

```
void NavegadorHistorial::VisitarURL(string url) {
    if (HistorialVacio()) {
        cima = new nodoP(url);
    } else {
        cima = new nodoP(url, cima);
```

```
    }
    cout << "Visitando: " << url << endl;
}

void NavegadorHistorial::RegresarPagina() {
    if (!HistorialVacio()) {
        pnodoP aux = cima;
        cout << "Regresando desde: " << aux->url << endl;
        cima = cima->siguiente;
        delete aux;
        if (cima) {
            cout << "Ahora en: " << cima->url << endl;
        } else {
            cout << "Historial vac o" << endl;
        }
    } else {
        cout << "No hay p ginas para regresar" << endl;
    }
}
```

7.3 Ejemplo: Cola de Impresión (Cola)

```
void ImpresoraQueue::AgregarDocumento(string documento) {
    if (ColaVacia()) {
        primero = new nodoC(documento);
        ultimo = primero;
    } else {
        ultimo->siguiente = new nodoC(documento);
        ultimo = ultimo->siguiente;
    }
    cout << "Documento '" << documento << "' a adido a la
    cola" << endl;
}

void ImpresoraQueue::ImprimirDocumento() {
    if (!ColaVacia()) {
        pnodoC aux = primero;
        cout << "Imprimiendo: " << aux->documento << endl;
        primero = primero->siguiente;
        if (!primero) ultimo = nullptr; // La cola qued
    vac a
        delete aux;
    } else {
        cout << "No hay documentos en la cola de impresi n"
     << endl;
    }
}
```

8 Consejos Prácticos para el Diseño de Estructuras de Datos

- **Encapsulamiento**: Mantén privados los atributos internos y expón solo las operaciones necesarias
- **Manejo de memoria**: Implementa destructores adecuados para evitar fugas de memoria
- **Validación**: Verifica siempre estados como listas vacías antes de realizar operaciones
- **Consistencia**: Mantén actualizados todos los punteros relevantes (primero, último, actual, etc.)
- **Simplicidad**: Diseña interfaces intuitivas que reflejen el comportamiento esperado

9 Conclusión

Hemos explorado tres estructuras de datos fundamentales, cada una con características y aplicaciones específicas:

- **Listas Doblemente Enlazadas**: Ideales para navegación bidireccional, como en reproductores multimedia
- **Pilas**: Perfectas para seguimiento de historial y operaciones que requieren deshacer
- **Colas**: Óptimas para procesamiento secuencial y gestión de recursos compartidos

La elección de la estructura de datos adecuada es crucial para el rendimiento y la claridad del código. Cada estructura tiene sus fortalezas y debilidades, y entender sus propiedades nos permite diseñar soluciones más eficientes y mantenibles.

10 Links Recomendados

- Visualización interactiva de estructuras de datos
- Simulador de estructuras de datos y algoritmos
- Tutorial completo sobre listas doblemente enlazadas
- Explicación detallada de pilas
- Guía práctica sobre colas

- Referencia de C++ oficial
- Tutoriales interactivos de C++