Sesión #1 de Tutorías de Estructuras de Datos

Tutor: Samir Cabrera Tabash

5 de marzo del 2025

1 Proceso de Compilación de C++

Antes de adentrarnos en las estructuras de datos, es fundamental comprender el proceso de compilación de C++, que transforma nuestro código fuente en un programa ejecutable:

1.1 Preprocesamiento

- **Función**: Procesa las directivas que comienzan con `#` (como `#include`, `#define`, `#ifdef`).
- **Proceso**:
  - Incluye el contenido de los archivos de cabecera (`.h`) en el código fuente.
  - Sustituye las macros definidas con `#define`.
  - Elimina comentarios del código.
  - Evalúa las directivas condicionales (`#ifdef`, `#ifndef`, etc.).
- **Resultado**: Un archivo de código fuente expandido y preparado para la compilación.

1.2 Compilación

- **Función**: Traduce el código preprocesado a lenguaje ensamblador.
- **Proceso**:
  - Analiza la sintaxis del código.
  - Realiza optimizaciones del código.
  - Genera código específico para la arquitectura objetivo.
- **Resultado**: Archivos en lenguaje ensamblador (`.s`).

1.3 Ensamblado

- **Función**: Convierte el código ensamblador en código objeto binario.
- **Proceso**:
  - Traduce las instrucciones de ensamblador a código máquina.
  - Crea tablas de símbolos para referencias entre archivos.
- **Resultado**: Archivos objeto (`.o` en sistemas Unix/Linux, `.obj` en Windows).

1.4 Enlazado (Linking)

- **Función**: Combina múltiples archivos objeto y bibliotecas en un único ejecutable.
- **Proceso**:
  - Resuelve las referencias entre diferentes archivos objeto.
  - Incluye el código de las bibliotecas utilizadas.
  - Organiza las secciones de código, datos y otros recursos.
- **Resultado**: Un archivo ejecutable (sin extensión en Linux/Unix, `.exe` en Windows).

2 Tipos de Archivos en Proyectos C++

2.1 Archivos `.h` (Header o Cabecera)

- **Propósito**: Contienen declaraciones de clases, funciones, variables y constantes.
- **Contenido típico**:
  - Declaraciones de clases y sus métodos.
  - Prototipos de funciones.
  - Definiciones de constantes y macros.
  - Declaraciones de variables externas.
- **Función en el proyecto**:
  - Permiten la modularización del código.
  - Facilitan la reutilización de código entre diferentes archivos.
  - Proporcionan interfaces para los módulos de código.

2.2 Archivos `.cpp` (Código Fuente)

- **Propósito**: Contienen la implementación del código.
- **Contenido típico**:
  - Definiciones de funciones declaradas en los archivos `.h`.
  - Implementación de los métodos de las clases.
  - Código ejecutable del programa.
- **Función en el proyecto**:
  - Implementan la lógica y funcionalidad del programa.
  - Se compilan independientemente en archivos objeto.

2.3 Archivos `.o` / `.obj` (Objetos)

- **Propósito**: Son el resultado intermedio de la compilación.
- **Contenido**: Código máquina sin enlazar.
- **Función en el proyecto**:
  - Representan unidades compiladas que aún no están enlazadas.
  - Permiten la compilación incremental (solo recompilar lo modificado).

2.4 Archivos `.lib` / `.a` (Bibliotecas)

- **Propósito**: Colecciones de código objeto reutilizable.
- **Tipos**:
  - **Bibliotecas estáticas** (`.lib` en Windows, `.a` en Unix/Linux): Se incorporan directamente en el ejecutable durante el enlazado.
  - **Bibliotecas dinámicas** (`.dll` en Windows, `.so` en Unix/Linux): Se cargan en tiempo de ejecución.
- **Función en el proyecto**:
  - Proporcionan funcionalidad común que puede ser utilizada por múltiples programas.
  - Reducen el tamaño del código fuente al separar la funcionalidad reutilizable.

3 Introducción a las Estructuras de Datos

Las listas enlazadas son estructuras de datos fundamentales en la programación. En esta sesión, exploraremos su implementación en C++, diferenciando entre listas simplemente enlazadas y listas circulares.

4 Definición de la Clase `nodo`

Cada nodo de la lista contiene un valor y un puntero al siguiente nodo. Se define de la siguiente manera:

```
class nodo {
public:
    nodo(int v);
    nodo(int v, nodo* signodo);
private:
    int valor;
    nodo* siguiente;
    friend class lista;
};
```

4.1 Explicación

- **Atributos privados**:
  - `int valor`: Este campo almacena el dato principal del nodo (en este caso un entero). En implementaciones más versátiles, podría ser reemplazado por un tipo genérico usando templates o por una estructura/clase que contenga múltiples campos. La encapsulación como atributo privado evita modificaciones no controladas desde fuera de las clases autorizadas.
  - `nodo* siguiente`: Puntero que almacena la dirección de memoria del siguiente nodo en la secuencia. Este puntero es la pieza clave que permite la navegación entre nodos sin necesidad de tenerlos contiguos en memoria.
    - En una lista simple, el último nodo tiene `siguiente = nullptr`.
    - En una lista circular, el `siguiente` del último nodo apunta al primer nodo (al nodo inicial).

    La manipulación incorrecta de este puntero puede causar pérdidas de memoria, ciclos infinitos o acceso a memoria inválida.
- **Métodos públicos**:
  - Constructores para inicializar los valores.
- **Mecanismo Friend**:
  - `friend class lista`: Declara a la clase `lista` como "amiga" de la clase `nodo`. Esta declaración es crucial para el diseño de la estructura de datos, ya que:
    - Mantiene la encapsulación (los atributos siguen siendo privados).
    - Permite que los métodos de `lista` manipulen directamente `valor` y `siguiente`.

* Evita la necesidad de crear métodos `getters/setters` que expondrían innecesariamente la implementación.

Sin esta declaración, la clase `lista` no podría modificar los enlaces entre nodos ni acceder a los valores.

5 Clase `lista` (Lista Simple)

Esta clase maneja la estructura de la lista enlazada:

```
class lista {
public:
    lista();
    ~lista();
    void InsertarInicio(int v);
    void InsertarFinal(int v);
    void InsertarPos(int v, int pos);
    bool ListaVacia();
    void Mostrar();
    void BorrarFinal();
    void BorrarInicio();
    void borrarPosicion(int pos);
    int largoLista();
private:
    pnodo primero;
};
```

5.1 Explicación

- **Atributos privados**: `primero` apunta al primer nodo.
- **Métodos públicos**: permiten insertar, eliminar y recorrer la lista.

6 Clase `listaC` (Lista Circular)

La lista circular es similar, pero el último nodo apunta al primero.

```
class listaC {
public:
    listaC();
    ~listaC();
    void InsertarInicio(int v);
    void InsertarFinal(int v);
    void InsertarPos(int v, int pos);
    bool ListaVacia();
    void Mostrar();
    void BorrarFinal();
    void BorrarInicio();
    void borrarPosicion(int pos);
```

```
    int largoLista();
private:
    pnodo primero;
};
```

7 Comparación entre Listas

Característica	Lista Simple	Lista Circular
Último nodo apunta a	`NULL`	Primer nodo
Fin de la lista	`NULL` encontrado	Se detecta el inicio
Recorrido completo	Una sola vez	Puede ser infinito
Implementación	Más simple	Requiere verificaciones adicionales

8 Implementación de Métodos Clave

8.1 Constructor y Destructor de Lista Simple

```
lista::lista() {
    primero = NULL;
}

lista::~lista() {
    pnodo aux;
    while(primero) {
        aux = primero;
        primero = primero->siguiente;
        delete aux;
    }
    primero = NULL;
}
```

8.2 Implementación de InsertarInicio

```
void lista::InsertarInicio(int v) {
    if (ListaVacia()) {
        primero = new nodo(v);
    } else {
        primero = new nodo(v, primero);
    }
}
```

8.3 Implementación de InsertarFinal (Lista Simple)

```
void lista::InsertarFinal(int v) {
    if (ListaVacia()) {
        primero = new nodo(v);
    } else {
        pnodo aux = primero;
        while (aux->siguiente) {
            aux = aux->siguiente;
        }
        aux->siguiente = new nodo(v);
    }
}
```

8.4 Implementación de InsertarFinal (Lista Circular)

```
void listaC::InsertarFinal(int v) {
    if (ListaVacia()) {
        primero = new nodo(v);
        primero->siguiente = primero; // Apunta a s  mismo
    } else {
        pnodo aux = primero;
        while (aux->siguiente != primero) {
            aux = aux->siguiente;
        }
        aux->siguiente = new nodo(v);
        aux->siguiente->siguiente = primero; // Completa el
    c rculo
    }
}
```

9 Uso en el `main.cpp`

Ejemplo de uso:

```
int main() {
    lista ListaSimple;
    ListaSimple.InsertarInicio(4);
    ListaSimple.InsertarInicio(56);
    ListaSimple.InsertarInicio(79);
    ListaSimple.InsertarFinal(80);
    ListaSimple.Mostrar();

    listaC ListaCircular;
    ListaCircular.InsertarInicio(4);
    ListaCircular.InsertarInicio(56);
    ListaCircular.InsertarInicio(79);
    ListaCircular.InsertarFinal(80);
    ListaCircular.Mostrar();

```

```
    return 0;
}
```

10 Flujo de Compilación de un Proyecto de Listas Enlazadas

Para ilustrar el proceso completo, consideremos un proyecto con los siguientes archivos:

- `nodo.h`: Contiene la declaración de la clase `nodo`
- `lista.h`: Contiene la declaración de las clases `lista` y `listaC`
- `nodo.cpp`: Implementa los métodos de la clase `nodo`
- `lista.cpp`: Implementa los métodos de la clase `lista`
- `listaC.cpp`: Implementa los métodos de la clase `listaC`
- `main.cpp`: Contiene la función principal

El proceso de compilación sería:

1. **Preprocesamiento**:
   - Se procesan todos los `#include` en `main.cpp`, `lista.cpp` y `listaC.cpp`
   - Se expanden todas las macros definidas
2. **Compilación**:
   - Se compila `nodo.cpp` a `nodo.o`
   - Se compila `lista.cpp` a `lista.o`
   - Se compila `listaC.cpp` a `listaC.o`
   - Se compila `main.cpp` a `main.o`
3. **Enlazado**:
   - Se enlazan `nodo.o`, `lista.o`, `listaC.o` y `main.o`
   - Se resuelven todas las referencias entre archivos
   - Se genera el ejecutable final

Un comando típico de compilación sería:

```
g++ -c nodo.cpp
g++ -c lista.cpp
g++ -c listaC.cpp
g++ -c main.cpp
g++ nodo.o lista.o listaC.o main.o -o programa
```

O de forma más compacta:

```
g++ nodo.cpp lista.cpp listaC.cpp main.cpp -o programa
```

11 Conclusión

Se ha demostrado el uso de listas enlazadas en C++, explicando sus métodos, encapsulación y diferencias entre listas simples y circulares. Estas estructuras son fundamentales para la programación eficiente. Además, se ha explicado el proceso completo de compilación de C++, desde el código fuente hasta el ejecutable final, pasando por el preprocesamiento, compilación, ensamblado y enlazado, así como la función de los diferentes tipos de archivos en un proyecto C++.

12 Links Recomendados

- Listas Circulares en otros lenguajes
- Visualizacion de listas linkeadas
- Visualizador de estructuras de datos
- Jetbrains Download
- Recomendacion Curso de git
- Ejemplo de funcionamiento de .cpp y .h
- Explicacion de tipos de archivos