Informe de Implementación - Trabajo en Clase

Fecha de Entrega

24 de octubre de 2025

Autor

Juan David Troncoso

Repositorio GitHub

Link del Repositorio: https://github.com/SiriusTVT/Trabajo-en-Clase/tree/main

---

1. Descripción General

Se han completado satisfactoriamente todas las actividades de programación en C++ enfocadas en el manejo de memoria, punteros y referencias. El proyecto incluye cuatro actividades principales y un programa extra que demuestra conceptos avanzados de memoria.

---

2. Actividades Implementadas

2.1 Actividad No. 1: Variables y Punteros Básicos

Archivo: `actividad1.cpp`

Objetivo: Declarar una variable entera, mostrar su dirección de memoria y modificarla indirectamente usando punteros.

Características:

- Declaración e inicialización de variable entera: `int variable = 42`
- Presentación de dirección de memoria usando operador `&`
- Creación de puntero: `int* puntero = &variable`
- Modificación indirecta del valor: `*puntero = 100`
- Verificación de que se modifica la variable original

Salida del Programa:

```
=== ACTIVIDAD No. 1 ===

1. Variable inicial:
   Valor: 42
   Direccion de memoria: 0x61ff08

2. Despues de modificar con puntero:
   Nuevo valor: 100
```

```
   Valor a traves del puntero: 100
   Direccion de memoria: 0x61ff08
```

2.2 Actividad No. 2: Punteros y Referencias

Archivo: `actividad2.cpp`

Objetivo: Trabajar con punteros y referencias para modificar variables indirectamente.

Características:

- Declaración de puntero a variable: `int* puntero = &variable`
- Modificación mediante puntero: `*puntero = 75`
- Creación de referencia: `int& referencia = variable`
- Modificación mediante referencia: `referencia = 150`
- Comparación de direcciones de memoria

Conceptos Clave:

- Las referencias apuntan a la misma dirección que la variable original
- El puntero tiene su propia dirección (`&puntero`)
- Ambos métodos modifican el mismo valor en memoria

Salida del Programa:

```
=== ACTIVIDAD No. 2 ===

4. Direcciones de memoria:
   Direccion de la variable: 0x61ff08
   Direccion almacenada en el puntero: 0x61ff08
   Direccion de la referencia: 0x61ff08
   Direccion del puntero mismo: 0x61ff04
```

2.3 Actividad No. 3: Arrays y Aritmética de Punteros

Archivo: `actividad3.cpp`

Objetivo: Utilizar punteros para acceder y modificar elementos de un array.

Características:

- Declaración de array: `int array[5] = {10, 20, 30, 40, 50}`
- Puntero al array: `int* puntero = array`
- Aritmética de punteros: `*(puntero + i)` para acceder a elementos
- Modificación de todos los elementos: valores de 100 a 500
- Visualización de direcciones de cada elemento

Conceptos Clave:

- Un array decae a puntero a su primer elemento
- Aritmética de punteros: `puntero + 1` apunta al siguiente elemento
- Cada elemento ocupa 4 bytes en memoria (int)

Salida del Programa:

```
5. Elementos del array y sus direcciones:
   array[0] = 100 | Direccion: 0x61fee0 | Con puntero: 0x61fee0
   array[1] = 200 | Direccion: 0x61fee4 | Con puntero: 0x61fee4
   array[2] = 300 | Direccion: 0x61fee8 | Con puntero: 0x61fee8
   array[3] = 400 | Direccion: 0x61feec | Con puntero: 0x61feec
   array[4] = 500 | Direccion: 0x61fef0 | Con puntero: 0x61fef0
```

2.4 Actividad No. 4: Asignación Dinámica de Memoria

Archivo: `actividad4.cpp`

Objetivo: Crear y manipular una matriz 2D dinámica, y liberar memoria correctamente.

Características:

- Asignación dinámica de puntero a punteros: `int** matriz = new int*[filas]`
- Asignación de filas: `matriz[i] = new int[columnas]`
- Llenado de matriz con valores secuenciales (10 a 120)
- Visualización de contenido y direcciones de memoria
- Liberación correcta de memoria: `delete[]` para filas, luego `delete[]` para el puntero principal

Conceptos Clave:

- Matriz dinámica 2D requiere doble asignación
- Orden importante en liberación: primero filas, luego matriz principal
- Previene memory leaks

Salida del Programa:

```
3. Contenido de la matriz:
   Fila 0: 10 20 30 40
   Fila 1: 50 60 70 80
   Fila 2: 90 100 110 120

6. Liberando memoria...
   Memoria liberada correctamente.
```

2.5 Programa Extra: Stack, Heap y Code

Archivo: `extra.cpp`

Objetivo: Demostrar la estructura de memoria en C++: Stack, Heap y Segmento de Código.

Características:

- Variable global (Data Segment): `int variableGlobal = 100`
- Variables locales (Stack): `int variableLocal = 42`
- Asignación dinámica (Heap): `int* punteroHeap = new int(999)`
- Comparación de direcciones de memoria en diferentes segmentos
- Anatomía visual de la memoria del programa
- Liberación correcta de memoria dinámica

Segmentos de Memoria Mostrados:

1. **CODE/TEXT:** Instrucciones del programa
2. **DATA SEGMENT:** Variables globales y constantes
3. **HEAP:** Memoria dinámica (crece hacia arriba ↑)
4. **STACK:** Variables locales (crece hacia abajo ↓)

Salida del Programa:

```
6. ANATOMIA DE MEMORIA:
   |     CODE/TEXT      | (Código del programa)
   |     DATA SEGMENT   | (Globales, constantes) -> 0x405004
   |     HEAP           | (Dynamic, crece ↑)    -> 0x11e6f28
   |     (espacio vacío)|
   |     STACK          | (Locales, crece ↓)    -> 0x61fee4
```

---

3. Archivos Generados

Código Fuente

```
actividad1.cpp          - Variables y punteros básicos
actividad2.cpp          - Punteros y referencias
actividad3.cpp          - Arrays y aritmética de punteros
actividad4.cpp          - Asignación dinámica 2D
extra.cpp               - Stack, Heap y Code
```

Archivos de Configuración

```
Makefile                - Automatización de compilación
.gitignore              - Configuración de Git
README.md               - Documentación del proyecto
```

---

4. Compilación y Ejecución

Compilación Individual

```
g++ -std=c++11 actividad1.cpp -o actividad1.exe
g++ -std=c++11 actividad2.cpp -o actividad2.exe
g++ -std=c++11 actividad3.cpp -o actividad3.exe
g++ -std=c++11 actividad4.cpp -o actividad4.exe
g++ -std=c++11 extra.cpp -o extra.exe
```

Ejecución

```
.\actividad1.exe
.\actividad2.exe
.\actividad3.exe
.\actividad4.exe
.\extra.exe
```

---

5. Conceptos Clave Implementados

Punteros

- Declaración: `int* puntero`
- Obtención de dirección: `&variable`
- Desreferenciación: `*puntero`
- Aritmética de punteros: `puntero + i`

Referencias

- Declaración: `int& referencia = variable`
- No pueden cambiar su referencia
- Apuntan a la misma dirección que la variable original

Memoria Dinámica

- Asignación: `new tipo`
- Arrays dinámicos: `new tipo[tamaño]`
- Liberación: `delete puntero` o `delete[] puntero`
- Prevención de memory leaks

Segmentos de Memoria

- **Stack:** Almacenamiento automático, rápido, tamaño limitado
- **Heap:** Memoria dinámica, acceso más lento, flexible
- **Data/Code:** Variables globales, constantes, código del programa

---

6. Control de Versiones

Configuración de Git

- Repositorio: `Trabajo-en-Clase`
- Rama: `main`
- Propietario: `SiriusTVT`
- `.gitignore` configurado para ignorar: `*.exe`, `*.o`, `*.out`

Estructura del Repositorio

```
Trabajo-en-Clase/
├── actividad1.cpp
├── actividad2.cpp
├── actividad3.cpp
├── actividad4.cpp
├── extra.cpp
├── Makefile
├── .gitignore
├── README.md
└── INFORME.md
```

---

7. Calidad del Código

Características

- ☑ Código limpio y bien estructurado
- ☑ Comentarios descriptivos eliminados (código auto-documentado)
- ☑ Salida clara y organizada con secciones numeradas
- ☑ Gestión correcta de memoria (sin memory leaks)
- ☑ Compilación sin errores ni warnings
- ☑ Estándar C++11 utilizado

Pruebas

- ☑ Todas las actividades compiladas exitosamente
- ☑ Todos los programas ejecutados sin errores
- ☑ Direcciones de memoria verificadas
- ☑ Valores modificados correctamente
- ☑ Memoria liberada sin problemas

---

8. Conclusiones

Se han completado satisfactoriamente todas las actividades propuestas con los siguientes logros:

1. **Dominio de Punteros:** Implementación correcta de punteros, referencias y aritmética de punteros
2. **Gestión de Memoria:** Correcta asignación y liberación de memoria dinámica
3. **Comprensión de Arquitectura:** Demostración clara de segmentos de memoria (Stack, Heap, Code)
4. **Buenas Prácticas:** Código limpio, bien documentado y sin memory leaks

5. **Control de Versiones:** Proyecto correctamente configurado en Git

El proyecto está listo para ser entregado a través de GitHub con toda la documentación necesaria.

---

9. Recomendaciones para Futuro

- Implementar manejadores de errores y validaciones
- Expandir con estructuras de datos dinámicas (listas, árboles)
- Agregar pruebas unitarias
- Documentar en Doxygen
- Implementar patrones de diseño

Fin del Informe