Práctica de Punteros en C++

Algoritmos y Estructuras de Datos II
Escuela de Ingeniería en Computadores
Tecnológico de Costa Rica

1. Introducción

En la programación en C++, los punteros son uno de los conceptos fundamentales que distinguen este lenguaje de muchos otros lenguajes de alto nivel. Mientras que algunos desarrolladores los consideran complejos, dominarlos es esencial para explotar todo el potencial que C++ tiene para ofrecer.

Un puntero es esencialmente una variable que almacena la dirección de memoria de otra variable, permitiendo manipular directamente la memoria del sistema. Esta característica otorga a C++ una notable flexibilidad y potencia, especialmente en:

- **Desarrollo de estructuras de datos dinámicas** - Listas enlazadas, árboles, grafos
- **Optimización de rendimiento** - Manipulación eficiente de grandes volúmenes de datos
- **Comunicación entre funciones** - Transferencia de referencias en lugar de copias completas
- **Polimorfismo y herencia** - Base para la programación orientada a objetos

Nota: Los punteros son la base para conceptos avanzados como la gestión dinámica de memoria, estructuras de datos enlazadas y polimorfismo, elementos fundamentales en el desarrollo de software moderno.

Consejo: Antes de comenzar con los ejercicios prácticos, asegúrese de comprender claramente:

- La diferencia entre un valor y una dirección de memoria
- El operador de dirección (&) y su uso para obtener la dirección de una variable
- El operador de indirección (*) y cómo se utiliza para acceder al valor apuntado
- La aritmética básica de punteros

Ejercicios Resueltos: Puede encontrar las soluciones a estos ejercicios en el siguiente repositorio de GitHub: https://github.com/stevmzz/practiceDatos2

2. Ejercicios

Ejercicio 1: Intercambio de Variables

Objetivo: Implementar una función que intercambie los valores de dos variables utilizando punteros, demostrando cómo se pueden modificar valores a través de direcciones de memoria.

Implemente una función `intercambiar(int* a, int* b)` que intercambie los valores de dos variables utilizando punteros. La función no debe retornar ningún valor, sino modificar directamente las variables originales.

```
void intercambiar(int* a, int* b) {
    // Completar esta funcion
}

int main() {
    int x = 5, y = 10;
    cout << "Antes: x = " << x << ", y = " << y << endl;
    intercambiar(&x, &y);
    cout << "Despues: x = " << x << ", y = " << y << endl;
    return 0;
}
```

Salida esperada:

```
Antes: x = 5, y = 10
Despues: x = 10, y = 5
```

Consejo: Para resolver este ejercicio, recuerde que los punteros permiten acceder y modificar directamente el valor de una variable mediante el operador de indirección (*). Utilice una variable temporal para almacenar uno de los valores durante el intercambio.

Video complementario | Punteros

Ejercicio 2: Búsqueda en Array

Objetivo: Implementar una función que busque un valor en un array y retorne un puntero al elemento encontrado, demostrando la utilidad de retornar punteros como resultado de operaciones.

Implemente una función `buscar(int* arr, int tamano, int valor)` que reciba un array, su tamaño y un valor a buscar, y retorne un puntero al primer elemento que coincida con el valor buscado. Si el valor no se encuentra, la función debe retornar NULL.

```
int* buscar(int* arr, int tamano, int valor) {
    // Completar esta funcion
}

int main() {
    int numeros[] = {10, 25, 30, 15, 40};
    int valorBuscado = 30;
    int* resultado = buscar(numeros, 5, valorBuscado);

    if (resultado != NULL) {
        cout << "Valor encontrado: " << *resultado << endl;
        cout << "Posicion: " << resultado - numeros << endl;
    } else {
        cout << "Valor no encontrado" << endl;
    }

    return 0;
}
```

Salida esperada:

```
Valor encontrado: 30
Posicion: 2
```

Nota: En este ejercicio, puede utilizar aritmética de punteros para recorrer el array. La expresión `resultado - numeros` calcula la diferencia en posiciones (no en bytes) entre dos punteros del mismo tipo, lo que nos permite determinar el índice del elemento encontrado.

Video complementario | Arreglos y Punteros

Ejercicio 3: Array Dinámico

Objetivo: Implementar funciones para la creación y gestión de memoria dinámica, demostrando el ciclo completo de asignación y liberación de memoria.

Implemente una función `crearArrayDinamico(int tamano)` que asigne memoria dinámicamente para un array de enteros del tamaño especificado, lo inicialice con valores aleatorios entre 1 y 100, y retorne un puntero al primer elemento. Además, implemente una función `liberarArray(int* arr)` para liberar la memoria asignada.

```
int* crearArrayDinamico(int tamano) {
    // Completar esta funcion
}

void liberarArray(int* arr) {
    // Completar esta funcion
}

int main() {
    int tamano = 5;
    int* miArray = crearArrayDinamico(tamano);

    cout << "Elementos del array: ";
    for (int i = 0; i < tamano; i++) {
        cout << miArray[i] << " ";
    }
    cout << endl;

    liberarArray(miArray);
    return 0;
}
```

Salida esperada (ejemplo):

```
Elementos del array: 42 87 15 36 91
```

Nota: Este ejercicio muestra cómo gestionar memoria dinámica para arrays de una dimensión. Es crucial inicializar los valores después de asignar memoria para evitar comportamientos impredecibles debido a valores residuales aleatorios en memoria. Utilice `new` para asignar memoria dinámica y `delete[]` para liberarla correctamente. La no liberación de memoria produce "memory leaks"que pueden degradar el rendimiento de su programa con el tiempo. Recuerde que tras cada operación `new` siempre debe existir una operación `delete[]` correspondiente.

Video complementario | Array Dinámico

Ejercicio 4: Matriz Dinámica

Objetivo: Implementar una matriz dinámica bidimensional utilizando punteros dobles, demostrando la gestión de estructuras de datos multidimensionales en memoria.

```
// Completar las siguientes funciones:

int** crearMatriz(int filas, int columnas) {
}
void llenarMatriz(int** matriz, int filas, int columnas) {
}
void mostrarMatriz(int** matriz, int filas, int columnas) {
}
void liberarMatriz(int** matriz, int filas) {
}

int main() {
    int filas = 3, columnas = 4;
    int** matriz = crearMatriz(filas, columnas);

    llenarMatriz(matriz, filas, columnas);
    cout << "Matriz generada:" << endl;
    mostrarMatriz(matriz, filas, columnas);

    liberarMatriz(matriz, filas);
    return 0;
}
```

Salida esperada (ejemplo):

```
Matriz generada:
12 45 33 8
76 22 54 11
39 61 28 5
```

Nota: En este ejercicio, debe tener en cuenta que una matriz dinámica bidimensional en C++ se implementa como un array de punteros (un puntero a punteros). Primero se asigna memoria para el array de punteros (las filas), y luego para cada fila por separado. Es importante liberar la memoria en orden inverso: primero cada fila y después el array de punteros, para evitar fugas de memoria.

Video complementario | Matriz Dinámica

Ejercicio 5: Estructura con Punteros

Objetivo: Utilizar punteros para manipular estructuras de datos personalizadas, combinando tipos de datos abstractos con asignación dinámica de memoria.

```
struct Estudiante {
    string nombre;
    string id;
    float calificacion;
};

// Completar las siguientes funciones:
Estudiante* crearEstudiantes(int n) {
}
void ingresarDatos(Estudiante* estudiantes, int n) {
}
void mostrarEstudiantes(Estudiante* estudiantes, int n) {
}
Estudiante* mejorEstudiante(Estudiante* estudiantes, int n) {
}

int main() {
    int numEstudiantes = 3;
    Estudiante* grupo = crearEstudiantes(numEstudiantes);

    ingresarDatos(grupo, numEstudiantes);
    cout << "\nLista de estudiantes:" << endl;
    mostrarEstudiantes(grupo, numEstudiantes);

    Estudiante* mejor = mejorEstudiante(grupo, numEstudiantes);
    cout << "\nEstudiante con mejor calificacion:" << endl;
    cout << "Nombre: " << mejor->nombre << endl;
    cout << "ID: " << mejor->id << endl;
    cout << "Calificacion: " << mejor->calificacion << endl;

    delete[] grupo;
    return 0;
}
```

Nota: En este ejercicio, observe el uso del operador de flecha (->) para acceder a los campos de una estructura a través de un puntero, que es equivalente a (*puntero).campo pero más compacto.

Video complementario | Estructura Básica

Ejercicio 6: Lista Enlazada Simple

Objetivo: Implementar una lista enlazada simple, una estructura de datos fundamental que demuestra el uso práctico de punteros para crear colecciones dinámicas de elementos.

```
struct Nodo {
    int dato;
    Nodo* siguiente;
};

// Completar estas funciones:
void insertarAlInicio(Nodo** cabeza, int valor) {
}
void insertarAlFinal(Nodo** cabeza, int valor) {
}
void eliminarNodo(Nodo** cabeza, int valor) {
}
void mostrarLista(Nodo* cabeza) {
}
void liberarLista(Nodo** cabeza) {
}

int main() {
    Nodo* miLista = NULL;
    insertarAlFinal(&miLista, 5);
    insertarAlFinal(&miLista, 10);
    insertarAlFinal(&miLista, 20);
    cout << "Lista original: ";
    mostrarLista(miLista);
    eliminarNodo(&miLista, 20);
    cout << "Despues de eliminar 20: ";
    mostrarLista(miLista);
    liberarLista(&miLista);
    return 0;
}
```

Salida esperada:

```
Lista original: 5 -> 10 -> 20 -> NULL
Despues de eliminar 20: 5 -> 10 -> NULL
```

Nota: Este ejercicio demuestra una aplicación fundamental de los punteros: la creación de estructuras de datos dinámicas. Note el uso de punteros dobles (`Nodo**`) en algunas funciones para poder modificar el puntero cabeza original.

Ejercicio 7: Pila Implementada con Punteros

Objetivo: Implementar una pila (stack) utilizando punteros, demostrando cómo se pueden construir estructuras de datos abstractas sobre la base de punteros.

```
struct Nodo {
    int dato;
    Nodo* siguiente;
};

// Completar estas funciones:
void push(Nodo** tope, int valor) {
}
int pop(Nodo** tope) {
}
int peek(Nodo* tope) {
}
bool estaVacia(Nodo* tope) {
}

int main() {
    Nodo* pila = NULL;
    push(&pila, 20);
    push(&pila, 30);
    cout << "Tope de la pila: " << peek(pila) << endl;
    cout << "Elementos extraidos: ";
    while (!estaVacia(pila)) {
        cout << pop(&pila) << " ";
    }
    cout << endl;
    cout << "La pila esta vacia? " << (estaVacia(pila) ? "Si" : "No"
   ) << endl;
    return 0;
}
```

Salida esperada:

```
Tope de la pila: 30
Elementos extraidos: 30 20
La pila esta vacia? Si
```

Advertencia: Cuando implemente la función `pop()`, asegúrese de verificar si la pila está vacía antes de intentar extraer elementos, para evitar operaciones sobre punteros nulos que podrían causar errores en tiempo de ejecución.

Ejercicio 8: Cola con Punteros

Objetivo: Implementar una cola (queue) utilizando punteros, presentando otro tipo de estructura de datos abstracta con un comportamiento diferente al de la pila.

```
struct Nodo {
    int dato;
    Nodo* siguiente;
};

// Completar las siguientes funciones:
void encolar(Nodo** frente, Nodo** final, int valor) {
}
int desencolar(Nodo** frente, Nodo** final) {
}
int consultarFrente(Nodo* frente) {
}
bool estaVacia(Nodo* frente) {
}

int main() {
    Nodo* frente = NULL;
    Nodo* final = NULL;
    encolar(&frente, &final, 10);
    encolar(&frente, &final, 20);
    cout << "Frente de la cola: " << consultarFrente(frente) << endl
    ;
    cout << "Elementos extraidos: ";
    while (!estaVacia(frente)) {
        cout << desencolar(&frente, &final) << " ";
    }
    cout << "La cola esta vacia? " << (estaVacia(frente) ? "Si" : "
    No") << endl;
    return 0;
}
```

Salida esperada:

```
Frente de la cola: 10
Elementos extraidos: 10 20
La cola esta vacia? Si
```

Nota: En contraste con la pila, la cola sigue el principio FIFO. Note que se necesitan dos punteros (frente y final) para mantener una implementación eficiente de la cola.

Ejercicio 9: Árbol Binario de Búsqueda

Objetivo: Implementar un Árbol Binario de Búsqueda (ABB), demostrando una estructura de datos más compleja donde los punteros son esenciales para establecer relaciones jerárquicas entre nodos.

```
struct Nodo {
    int valor;
    Nodo* izquierdo;
    Nodo* derecho;
};

// Completar las siguientes funciones:
Nodo* crearNodo(int valor) {
}
Nodo* insertarNodo(Nodo* raiz, int valor) {
}
Nodo* buscarNodo(Nodo* raiz, int valor) {
}
void inOrden(Nodo* raiz) {
}
void liberarArbol(Nodo* raiz) {
}

int main() {
    Nodo* arbol = NULL;
    arbol = insertarNodo(arbol, 50);
    insertarNodo(arbol, 30);
    insertarNodo(arbol, 20);
    insertarNodo(arbol, 40);
    insertarNodo(arbol, 70);
    insertarNodo(arbol, 60);
    cout << "Recorrido en orden: ";
    inOrden(arbol);
    int valorBuscado = 40;
    Nodo* resultado = buscarNodo(arbol, valorBuscado);
    if (resultado) {
        cout << "Valor " << valorBuscado << " encontrado en el arbol
    " << endl;
    } else {
        cout << "Valor " << valorBuscado << " no encontrado" << endl
    ;
    }
    liberarArbol(arbol);
    return 0;
}
```

Salida esperada:

```
Recorrido en orden: 20 30 40 50 60 70
Valor 40 encontrado en el arbol
```

Nota: Este ejercicio demuestra una aplicación avanzada de punteros: la implementación de un árbol binario. Observa cómo la función `insertarNodo` devuelve un puntero actualizado a la raíz, lo que permite mantener la estructura correcta del árbol.

Consejo: La liberación de memoria en estructuras recursivas como los árboles se implementa eficientemente mediante un algoritmo recursivo de recorrido post-orden, donde primero se liberan los subárboles izquierdo y derecho, y finalmente el nodo actual.

Ejercicio 10: Punteros a Funciones

Objetivo: Utilizar punteros a funciones para implementar comportamientos parametrizables, demostrando cómo los punteros pueden referirse no solo a datos sino también a código ejecutable.

```
int duplicar(int n) {
    return n * 2;
}

int triplicar(int n) {
    return n * 3;
}

int alCuadrado(int n) {
    return n * n;
}

void aplicarOperacion(int* arr, int tamano, int (*operacion)(int)) {
    // Completar esta funcion
}

void mostrarArray(int* arr, int tamano) {
    for(int i = 0; i < tamano; i++) {
        cout << arr[i] << " ";
    }
    cout << endl;
}

int main() {
    int numeros[] = {1, 2, 3, 4, 5};
    int tamano = 5;
    cout << "Array original: ";
    mostrarArray(numeros, tamano);
    cout << "Despues de duplicar: ";
    aplicarOperacion(numeros, tamano, duplicar);
    mostrarArray(numeros, tamano);
    cout << "Despues de triplicar: ";
    aplicarOperacion(numeros, tamano, triplicar);
    mostrarArray(numeros, tamano);
    cout << "Despues de elevar al cuadrado: ";
    aplicarOperacion(numeros, tamano, alCuadrado);
    mostrarArray(numeros, tamano);
    return 0;
}
```

Salida esperada:

```
Array original: 1 2 3 4 5
Despues de duplicar: 2 4 6 8 10
Despues de triplicar: 6 12 18 24 30
Despues de elevar al cuadrado: 36 144 324 576 900
```

Nota: Este ejercicio ilustra el concepto avanzado de punteros a funciones, que permite pasar comportamiento como parámetro. Esto es similar al concepto de estrategias o callbacks en lenguajes de más alto nivel y constituye la base de técnicas de programación funcional en C++.