Resumen I Examen Datos

Alejandro León Marín

31 de julio de 2024

Índice

1. Variables y Arreglos

1.1. Enteros

Los tipos de datos enteros en C++ incluyen `int`, `short`, `long` y `long long`. Estos tipos se utilizan para almacenar números enteros, positivos y negativos.

1.1.1. Declaración e Inicialización

```
int a = 10;
short b = 5;
long c = 1234567890;
long long d = 123456789012345678 9LL;
```

1.1.2. Operaciones Básicas

Las operaciones básicas con enteros incluyen suma, resta, multiplicación, división y módulo.

```
int x = 10, y = 3;
int sum = x + y;        // Suma
int diff = x - y;       // Resta
int prod = x * y;       // Multiplicacion
int quot = x / y;       // Division
int mod = x % y;        // Modulo
```

1.2. Reales

Los tipos de datos de punto flotante en C++ incluyen `float`, `double` y `long double`. Estos tipos se utilizan para almacenar números reales con decimales.

1.2.1. Declaración e Inicialización

```
float pi = 3.14159f;
double e = 2.718281828459045;
long double bigNumber = 3.14159265358979323846264338327 9L;
```

1.2.2. Operaciones Básicas

Las operaciones básicas con números reales incluyen suma, resta, multiplicación, división y comparación.

```
float a = 1.5f, b = 2.5f;
float sum = a + b;        // Suma
float diff = a - b;       // Resta
float prod = a * b;       // Multiplicacion
```

```
float quot = a / b;       // Division
bool isEqual = (a == b); // Comparacion
```

1.3. Caracteres

El tipo de datos `char` se utiliza para almacenar caracteres individuales. También se pueden utilizar para representar cadenas de caracteres (`std::string`).

1.3.1. Declaración e Inicialización

```
char letter = 'A';
std::string greeting = "Hello, World!";
```

1.3.2. Operaciones Básicas

Las operaciones básicas con caracteres incluyen asignación, concatenación y comparación.

```
char ch1 = 'A', ch2 = 'B';
bool isEqual = (ch1 == ch2); // Comparacion

std::string str1 = "Hello";
std::string str2 = "World";
std::string str3 = str1 + ", " + str2 + "!"; // Concatenacion
```

1.4. Archivos

La manipulación de archivos en C++ se realiza mediante las bibliotecas `<fstream>`, `<ifstream>` y `<ofstream>`. Estas permiten leer y escribir archivos.

1.4.1. Lectura de Archivos

```
#include <fstream>
#include <iostream>
#include <string>

int main() {
    std::ifstream inFile("input.txt");
    std::string line;
    if (inFile.is_open()) {
        while (std::getline(inFile, line)) {
            std::cout << line << std::endl;
        }
        inFile.close();
    } else {
```

```
        std::cerr << "Unable to open file";
    }
    return 0;
}
```

1.4.2. Escritura de Archivos

```
#include <fstream>
#include <iostream>

int main() {
    std::ofstream outFile("output.txt");
    if (outFile.is_open()) {
        outFile << "Hello, file!" << std::endl;
        outFile.close();
    } else {
        std::cerr << "Unable to open file";
    }
    return 0;
}
```

1.5. Arreglos

Los arreglos en C++ se utilizan para almacenar colecciones de elementos del mismo tipo. Pueden ser de tamaño fijo o dinámico (utilizando `std::vector`).

1.5.1. Arreglos de Tamaño Fijo

```
int fixedArray[5] = {1, 2, 3, 4, 5};

// Acceso a elementos
int firstElement = fixedArray[0];
int thirdElement = fixedArray[2];
```

1.5.2. Vectores (Arreglos Dinámicos)

```
#include <vector>
#include <iostream>

int main() {
    std::vector<int> dynamicArray = {1, 2, 3, 4, 5};
    dynamicArray.push_back(6); // Aniadir un elemento al final

    // Acceso a elementos
```

```
    int firstElement = dynamicArray[0];
    int thirdElement = dynamicArray[2];

    // Iterar sobre el vector
    for (int i : dynamicArray) {
        std::cout << i << " ";
    }
    std::cout << std::endl;

    return 0;
}
```

1.6. Arreglos Multidimensionales

```
int matrix[3][3] = {
    {1, 2, 3},
    {4, 5, 6},
    {7, 8, 9}
};

// Acceso a elementos
int element = matrix[1][2]; // Elemento en la segunda fila, tercera columna
```

2. Punteros

Los punteros en C++ son herramientas poderosas que permiten manipular directamente las direcciones de memoria. Esto facilita el manejo eficiente de datos, ya que podemos acceder y modificar valores sin necesidad de copiar datos innecesariamente.

2.1. Declaración de Punteros

Para declarar un puntero, se usa el operador `*` junto con el tipo de dato al que el puntero apuntará.

```
int *puntero;
```

En este ejemplo, `puntero` es un puntero a un entero. Para asignar la dirección de memoria de una variable al puntero, usamos el operador `&`.

```
int variable = 5;
int *puntero = &variable;
```

2.2. Acceso al Valor de un Puntero

Para acceder al valor en la dirección de memoria a la que apunta un puntero, se utiliza el operador `*` (desreferenciación).

```
int a = 10;
int* p = &a;
int b = *p; // b es 10, el valor de a
*p = 20; // a ahora es 20
```

2.3. Punteros y Arreglos

Los punteros pueden facilitar el acceso a los elementos de un arreglo.

```
int arreglo[5] = {1, 2, 3, 4, 5};
int* p = arreglo; // p apunta al primer elemento del arreglo
int a = *p; // a es 1
int b = *(p + 1); // b es 2
```

2.4. Punteros a Funciones

Los punteros también pueden apuntar a funciones, permitiendo llamadas indirectas a funciones.

```
int suma(int a, int b) {
    return a + b;
}
```

```
int (*puntero)(int, int) = suma;
int resultado = (*puntero)(2, 3); // resultado es 5
```

2.5. Punteros a Estructuras

Es posible apuntar a estructuras y acceder a sus miembros usando punteros.

```
struct Persona {
    std::string nombre;
    int edad;
};

Persona p = {"Juan", 20};
Persona* puntero = &p;
std::string nombre = puntero->nombre; // nombre es "Juan"
```

2.6. Punteros a Punteros

Los punteros pueden apuntar a otros punteros, formando niveles adicionales de indirectividad.

```
int a = 10;
int* p = &a;
int** pp = &p;
int b = **pp; // b es 10
```

2.7. Punteros a Constantes

Un puntero puede apuntar a una constante, lo que impide modificar el valor apuntado a través del puntero.

```
const double PI = 3.14159;
const double* p = &PI;
double pi = *p; // pi es 3.14159
```

2.8. Punteros a Funciones Miembro

Los punteros pueden apuntar a funciones miembro de una clase, y se utilizan con una sintaxis especial.

```
class Persona {
public:
    void saludar() {
        std::cout << "Hola" << std::endl;
    }
};
```

```
void (Persona::*puntero)() = &Persona::saludar;
Persona p;
(p.*puntero)(); // Imprime "Hola"
```

3. Listas en C++

Las listas en C++ son contenedores de la biblioteca estándar (STL) que permiten almacenar una colección de elementos de manera secuencial. A diferencia de los arreglos y vectores, las listas permiten inserciones y eliminaciones eficientes en cualquier posición, ya que están implementadas como listas doblemente enlazadas.

3.1. Declaración y Inicialización

Para utilizar listas, es necesario incluir el encabezado `<list>`.

```
#include <list>

std::list<int> numeros;
std::list<std::string> palabras = {"hola", "mundo"};
```

3.2. Operaciones Básicas

Las listas soportan una variedad de operaciones, incluyendo inserción, eliminación y acceso a elementos.

- **Inserción**: `push_back()` agrega un elemento al final, y `push_front()` al inicio.
- **Eliminación**: `pop_back()` elimina el último elemento, y `pop_front()` el primero.
- **Acceso**: Las listas no permiten acceso aleatorio, pero se pueden recorrer con iteradores.

```
numeros.push_back(5); // Agrega 5 al final
numeros.push_front(1); // Agrega 1 al inicio
numeros.pop_back(); // Elimina el ultimo elemento
numeros.pop_front(); // Elimina el primer elemento
```

3.3. Iteradores

Las listas utilizan iteradores para acceder a sus elementos. Un iterador es similar a un puntero, permitiendo moverse a través de la lista.

```
for (std::list<int>::iterator it = numeros.begin(); it != numeros.end(); ++i
    std::cout << *it << std::endl;  // Imprime cada elemento de la lista
}
```

3.4. Ordenamiento y Búsqueda

Las listas en C++ proporcionan métodos para ordenar y buscar elementos.

```
numeros.sort();  // Ordena los elementos
numeros.reverse();  // Invierte el orden de los elementos
```

3.5. Comparación con Otros Contenedores

Las listas son más lentas en acceso aleatorio que los vectores, pero más rápidas en inserciones y eliminaciones en el medio del contenedor. No es una estructura ideal para situaciones que requieren acceso rápido a elementos por índice, pero es excelente para escenarios donde la inserción y eliminación de elementos es frecuente.

3.6. Ejemplo Completo

```
#include <iostream>
#include <list>

int main() {
    std::list<int> lista = {1, 2, 3, 4, 5};
    lista.push_back(6);
    lista.sort();

    for (auto it = lista.begin(); it != lista.end(); ++it) {
        std::cout << *it << " ";
    }

    return 0;
}
```

Este código inicializa una lista, agrega un elemento, la ordena y luego imprime todos sus elementos.

3.7. Tipos de Listas

3.7.1. Listas Generales

Las listas generales son estructuras de datos que permiten almacenar elementos en una secuencia. Cada elemento puede estar enlazado a otros elementos de la lista. Existen varios tipos de listas generales, como listas simples, listas doblemente enlazadas y listas circulares.

3.7.2. Listas Simples

Las listas simples, también conocidas como listas simplemente enlazadas, son una forma básica de lista enlazada. En una lista simple, cada nodo contiene un

dato y un puntero al siguiente nodo en la secuencia. El último nodo de la lista apunta a `null`, indicando el final de la lista.

Características:

- Fácil de implementar.
- Inserción y eliminación de nodos es eficiente.
- Búsqueda de un nodo específico puede ser lenta si la lista es larga, ya que requiere recorrer los nodos secuencialmente.

Ejemplo:

```
Cabeza -> Nodo1 -> Nodo2 -> Nodo3 -> Null
```

3.7.3. Listas Circulares

Las listas circulares son un tipo de lista enlazada en la que el último nodo está conectado de nuevo al primer nodo, formando un círculo. Esto permite que la lista sea recorrida de manera cíclica.

Características:

- No hay un nodo que apunte a `null`.
- Puede ser útil para aplicaciones que requieren un bucle continuo, como en sistemas operativos para manejo de procesos.
- Al igual que las listas simples, la inserción y eliminación de nodos es eficiente.
- La búsqueda de un nodo específico puede ser lenta en listas largas.

Ejemplo:

```
Cabeza -> Nodo1 -> Nodo2 -> Nodo3 --+
^                                   |
|                                   |
+-----------------------------------+
```

3.8. Ejemplos

3.8.1. Lista Simple en C++

```
#include <iostream>

struct Node {
    int data;
    Node* next;
```

```
};

class SinglyLinkedList {
public:
    SinglyLinkedList() : head(nullptr) {}

    void insert(int value) {
        Node* newNode = new Node();
        newNode->data = value;
        newNode->next = head;
        head = newNode;
    }

    void display() const {
        Node* current = head;
        while (current != nullptr) {
            std::cout << current->data << " -> ";
            current = current->next;
        }
        std::cout << "null\n";
    }

    ~SinglyLinkedList() {
        while (head != nullptr) {
            Node* temp = head;
            head = head->next;
            delete temp;
        }
    }

private:
    Node* head;
};

int main() {
    SinglyLinkedList list;
    list.insert(3);
    list.insert(2);
    list.insert(1);
    list.display(); // Output: 1 -> 2 -> 3 -> null
    return 0;
}
```

3.8.2. Listas Doblemente Enlazadas en C++

```
#include <iostream>

struct Node {
    int data;
    Node* prev;
    Node* next;
};

class DoublyLinkedList {
public:
    DoublyLinkedList() : head(nullptr), tail(nullptr) {}

    void insert(int value) {
        Node* newNode = new Node();
        newNode->data = value;
        newNode->next = head;
        newNode->prev = nullptr;
        if (head != nullptr) {
            head->prev = newNode;
        } else {
            tail = newNode;
        }
        head = newNode;
    }

    void displayForward() const {
        Node* current = head;
        while (current != nullptr) {
            std::cout << current->data << " -> ";
            current = current->next;
        }
        std::cout << "null\n";
    }

    void displayBackward() const {
        Node* current = tail;
        while (current != nullptr) {
            std::cout << current->data << " -> ";
            current = current->prev;
        }
        std::cout << "null\n";
    }

    ~DoublyLinkedList() {
        while (head != nullptr) {
            Node* temp = head;
```

```
            head = head->next;
            delete temp;
        }
    }

private:
    Node* head;
    Node* tail;
};

int main() {
    DoublyLinkedList list;
    list.insert(3);
    list.insert(2);
    list.insert(1);
    list.displayForward(); // Output: 1 -> 2 -> 3 -> null
    list.displayBackward(); // Output: 3 -> 2 -> 1 -> null
    return 0;
}
```

3.8.3. Lista Circular en C++

```
#include <iostream>

struct Node {
    int data;
    Node* next;
};

class CircularLinkedList {
public:
    CircularLinkedList() : tail(nullptr) {}

    void insert(int value) {
        Node* newNode = new Node();
        newNode->data = value;
        if (tail == nullptr) {
            tail = newNode;
            tail->next = tail;
        } else {
            newNode->next = tail->next;
            tail->next = newNode;
            tail = newNode;
        }
    }
```

```
    void display() const {
        if (tail == nullptr) return;
        Node* current = tail->next;
        do {
            std::cout << current->data << " -> ";
            current = current->next;
        } while (current != tail->next);
        std::cout << "(back to head)\n";
    }

    ~CircularLinkedList() {
        if (tail != nullptr) {
            Node* current = tail->next;
            while (current != tail) {
                Node* temp = current;
                current = current->next;
                delete temp;
            }
            delete tail;
        }
    }

private:
    Node* tail;
};

int main() {
    CircularLinkedList list;
    list.insert(3);
    list.insert(2);
    list.insert(1);
    list.display(); // Output: 1 -> 2 -> 3 -> (back to head)
    return 0;
}
```

4. Pilas

5. Colas

6. Recursividad