VÕ TIẾN
Thảo luận kiến thức CNTT trường BK về KHMT(CScience), KTMT(CEngineering)
https://www.facebook.com/groups/khmt.ktmt.cse.bku

Cấu Trúc Dữ Liệu và Giải Thuật (DSA)

DSA1 - HK241

Developing List Data Structures and Artificial Neural Networks

Thảo luận kiến thức CNTT trường BK
về KHMT(CScience), KTMT(CEngineering)
https://www.facebook.com/groups/khmt.ktmt.cse.bku

TP. HỒ CHÍ MINH, THÁNG 9/2024

Mục lục

1 Kiến thức lập trình Cần Làm BTL

1.1 Kiến thức về con trỏ

1. **Cấu trúc cơ bản của con trỏ:**
   - **Khai báo con trỏ:**

```
int* ptr;  // Con trỏ kiểu int
```

   Ở đây, `ptr` là một con trỏ kiểu `int`, nghĩa là nó có thể chứa địa chỉ của một biến kiểu `int`.
   - **Gán địa chỉ cho con trỏ:** Con trỏ được gán địa chỉ của một biến bằng toán tử `&`.

```
int a = 10;
int* ptr = &a;  // ptr trỏ đến biến a
```

   - **Truy xuất giá trị mà con trỏ trỏ đến:** Để truy xuất giá trị của biến mà con trỏ trỏ tới, ta sử dụng toán tử `*` (dereference).

```
int value = *ptr;  // Lấy giá trị của biến a thông qua con trỏ ptr
```

2. **Các khái niệm quan trọng liên quan đến con trỏ:**
   - **Toán tử `&`:** Trả về địa chỉ của biến.

```
int a = 5;
int* ptr = &a;  // &a trả về địa chỉ của biến a
```

   - **Toán tử `*`:** Truy cập giá trị mà con trỏ trỏ tới (dereferencing).

```
int value = *ptr;  // Lấy giá trị của biến a thông qua ptr
```

   - **Con trỏ NULL:** Con trỏ không trỏ đến bất kỳ biến nào, thường được gán giá trị `nullptr`, `0`, hoặc `NULL`.

```
int* ptr = nullptr;
int* ptr = 0;
```

   - **Con trỏ hàm:** Con trỏ có thể trỏ tới một hàm, cho phép thực hiện các thao tác gọi hàm thông qua con trỏ.

```
        void (*funcPtr)() = &someFunction;
        (*funcPtr)();  // Gọi hàm thông qua con trỏ
```

   - **Con trỏ đến con trỏ:** Con trỏ có thể trỏ đến một con trỏ khác.

```
int a = 10;
int* ptr = &a;
int** ptr2 = &ptr;  // ptr2 trỏ đến ptr
```

- **Con trỏ động và quản lý bộ nhớ:**
  - Cấp phát bộ nhớ động bằng `new`.
  - Giải phóng bộ nhớ bằng `delete`.

```
int* ptr = new int;  // Cấp phát bộ nhớ động
delete ptr;          // Giải phóng bộ nhớ
```

- **Con trỏ và mảng:** Mảng và con trỏ có liên hệ chặt chẽ với nhau. Tên của mảng chính là địa chỉ của phần tử đầu tiên.

```
int arr[5] = {1, 2, 3, 4, 5};
int* ptr = arr;  // ptr trỏ đến phần tử đầu tiên của arr
```

3. **Toán tử ++ và - với con trỏ:** Toán tử ++ và - có thể được sử dụng để tăng hoặc giảm địa chỉ mà con trỏ trỏ đến, thường được dùng khi duyệt qua mảng.

```
int arr[3] = {10, 20, 30};
int* ptr = arr;

ptr++;  // Trỏ đến phần tử tiếp theo (arr[1])
ptr--;  // Trỏ lại phần tử trước đó (arr[0])
```

   Khi sử dụng toán tử ++ hoặc -, con trỏ sẽ được tăng hoặc giảm theo kích thước của kiểu dữ liệu mà nó trỏ đến. Ví dụ, với con trỏ kiểu `int`, mỗi lần tăng (`ptr++`), con trỏ sẽ nhảy qua 4 byte (vì kích thước của `int` là 4 byte).

4. **Các lỗi thường gặp khi làm việc với con trỏ:**
   - **Dereference con trỏ NULL:** Truy xuất giá trị từ một con trỏ không trỏ đến địa chỉ hợp lệ.
   - **Memory leak:** Không giải phóng bộ nhớ sau khi dùng `new`.
   - **Dangling pointer:** Con trỏ trỏ tới một vùng nhớ đã bị giải phóng.
5. **Toán tử & trong tham chiếu:** Ngoài việc sử dụng để lấy địa chỉ của một biến, toán tử `&` còn được sử dụng để khai báo biến tham chiếu trong C++. Một biến tham chiếu là một bí danh (alias) cho một biến khác, cho phép truy cập và thay đổi giá trị của biến đó thông qua tham chiếu.

   Ví dụ:

```
int a = 5;
int& ref = a;  // ref là tham chiếu đến biến a
ref = 10;      // Thay đổi giá trị của a thông qua ref
```

   Trong trường hợp này, `ref` là một tham chiếu đến `a`, và mọi thay đổi thông qua `ref` sẽ ảnh hưởng đến `a`.

1.2 Sự khác nhau giữa & và * khi truyền đối tượng XArrayList vào hàm

Trong C++, khi truyền một đối tượng vào hàm, có hai cách phổ biến: truyền bằng tham chiếu (dùng &) và truyền bằng con trỏ (dùng *). Dưới đây là sự khác nhau giữa hai cách này khi truyền đối tượng kiểu XArrayList.

1. Truyền bằng tham chiếu (&) Khi truyền đối tượng kiểu XArrayList vào hàm bằng tham chiếu, ta dùng toán tử &. Điều này giúp hàm làm việc trực tiếp với đối tượng gốc mà không cần tạo bản sao, và cú pháp bên trong hàm vẫn giống như khi làm việc với biến thông thường.

```
void modifyList(XArrayList& list) {
    // Thao tác trực tiếp trên đối tượng list
    list.add(10);
}
```

Đặc điểm:

- Không cần giải phóng bộ nhớ thủ công.
- Cú pháp dễ hiểu, giống như làm việc với biến thông thường.
- Không cần dereference (truy xuất giá trị) như với con trỏ.
- Đảm bảo rằng đối tượng không thể là nullptr (không thể trỏ tới địa chỉ không hợp lệ).

2. Truyền bằng con trỏ (*) Khi truyền đối tượng kiểu XArrayList vào hàm bằng con trỏ, ta dùng toán tử *. Trong trường hợp này, hàm nhận địa chỉ của đối tượng và thao tác trên đối tượng thông qua địa chỉ đó, do đó cần phải dereference để truy cập giá trị của đối tượng.

```
void modifyList(XArrayList* list) {
    if (list != nullptr) {
        // Thao tác trên đối tượng thông qua con trỏ
        list->add(10);
    }
}
```

Đặc điểm:

- Cần kiểm tra nếu con trỏ hợp lệ (không phải nullptr).
- Phải dereference con trỏ mỗi khi thao tác với đối tượng, dùng -> thay vì dấu ..
- Cho phép đối tượng truyền vào có thể là nullptr, từ đó không thực hiện bất kỳ thay đổi nào nếu không hợp lệ.
- Cần quản lý bộ nhớ cẩn thận hơn nếu đối tượng được cấp phát động.

3. Sự khác biệt chính

Đặc điểm	Tham chiếu (&)	Con trỏ (*)
Cú pháp	Dùng như đối tượng thông thường	Phải dereference để truy xuất
Kiểm tra hợp lệ	Không cần	Phải kiểm tra nếu con trỏ là nullptr
Quản lý bộ nhớ	Không cần	Có thể cần giải phóng nếu cấp phát động
Rủi ro	Không trỏ đến nullptr	Có thể trỏ tới địa chỉ không hợp lệ

2 Lỗi Thường Gặp Với Con Trỏ

2.1 Con Trỏ Null (Null Pointer)

Mô Tả: Sử dụng con trỏ mà không khởi tạo hoặc sử dụng con trỏ có giá trị `nullptr` có thể dẫn đến lỗi khi cố gắng truy cập bộ nhớ không hợp lệ.

Cách Tránh:

- **Khởi Tạo Con Trỏ:** Luôn khởi tạo con trỏ trước khi sử dụng. Đặt nó về `nullptr` nếu chưa có giá trị hợp lệ.

```
int* ptr = nullptr;  // Khởi tạo con trỏ
```

- **Kiểm Tra Null:** Trước khi sử dụng con trỏ, kiểm tra xem nó có phải là `nullptr` không.

```
if (ptr != nullptr) {
    *ptr = 5;  // Chỉ sử dụng ptr nếu nó không phải là nullptr
}
```

2.2 Rò Rỉ Bộ Nhớ (Memory Leak)

Mô Tả: Rò rỉ bộ nhớ xảy ra khi cấp phát bộ nhớ động mà không giải phóng nó, dẫn đến việc bộ nhớ không được giải phóng và không thể sử dụng lại.

Cách Tránh:

- **Giải Phóng Bộ Nhớ:** Luôn giải phóng bộ nhớ sau khi không còn sử dụng.

```
int* ptr = new int[10];
delete[] ptr;  // Giải phóng bộ nhớ
```

- **Theo Dõi Quản Lý Bộ Nhớ:** Đảm bảo mọi khu vực cấp phát bộ nhớ có một lệnh giải phóng tương ứng.

```
void allocateMemory() {
    int* ptr = new int[10];
    // Sử dụng ptr
    delete[] ptr;  // Giải phóng bộ nhớ sau khi sử dụng
}
```

2.3 Truy Cập Ngoài Phạm Vi (Out-of-Bounds Access)

Mô Tả: Truy cập đến vùng bộ nhớ ngoài phạm vi của mảng có thể dẫn đến lỗi nghiêm trọng, bao gồm ghi đè dữ liệu và lỗi thời gian chạy.

Cách Tránh:

- **Kiểm Tra Phạm Vi:** Đảm bảo rằng chỉ số bạn sử dụng để truy cập mảng hợp lệ.

```
int arr[5] = {1, 2, 3, 4, 5};
int index = 2;
if (index >= 0 && index < 5) {
    int value = arr[index];
}
```

2.4 Double Delete (Xóa Bộ Nhớ Nhiều Lần)

Mô Tả: Gọi `delete` hoặc `delete[]` nhiều lần trên cùng một vùng bộ nhớ dẫn đến lỗi nghiêm trọng, như lỗi phân đoạn (segmentation fault).

Cách Tránh:

- **Đặt Con Trỏ Thành `nullptr`:** Sau khi giải phóng bộ nhớ, đặt con trỏ về `nullptr` để tránh việc giải phóng lại bộ nhớ đó.

```
int* ptr = new int;
delete ptr;
ptr = nullptr;  // Đặt con trỏ thành nullptr
```

- **Kiểm Tra Trước Khi Giải Phóng:** Kiểm tra con trỏ trước khi giải phóng để đảm bảo nó không bị giải phóng nhiều lần.

```
if (ptr != nullptr) {
    delete ptr;
    ptr = nullptr;
}
```

2.5 Gán Con Trỏ Đến Vùng Bộ Nhớ Đã Giải Phóng

Mô Tả: Sử dụng con trỏ sau khi vùng bộ nhớ nó trỏ đến đã được giải phóng có thể dẫn đến hành vi không xác định.

Cách Tránh:

- **Đặt Con Trỏ Thành `nullptr`:** Ngay sau khi giải phóng bộ nhớ, đặt con trỏ về `nullptr`.

```
int* ptr = new int;
delete ptr;
ptr = nullptr;  // Tránh việc sử dụng con trỏ đã giải phóng
```

- **Cẩn Thận Khi Gán Con Trỏ:** Đảm bảo rằng không có con trỏ nào khác trỏ đến cùng một vùng bộ nhớ nếu bạn giải phóng nó.

```
int* ptr1 = new int;
int* ptr2 = ptr1;  // ptr2 cũng trỏ đến vùng bộ nhớ mà ptr1 trỏ đến
delete ptr1;       // ptr2 không còn hợp lệ
ptr1 = nullptr;
ptr2 = nullptr;   // Đặt ptr2 thành nullptr để tránh việc sử dụng không hợp lệ
```

3 Kiến thức về OOP cần làm BTL

3.1 Cơ bản của OOP

1. **Thuộc tính (Attribute):** Thuộc tính là các biến thành viên của một lớp, chứa thông tin hoặc trạng thái của đối tượng. Các thuộc tính được khai báo trong lớp và mỗi đối tượng của lớp có thể có các giá trị khác nhau cho những thuộc tính này.

```cpp
class Person {
private:
    std::string name;
    int age;

public:
    Person(std::string n, int a) : name(n), age(a) {}
};
```

2. **Phương thức (Method):** Phương thức là các hàm thành viên của lớp, được sử dụng để thực hiện các hành vi hoặc thao tác trên các thuộc tính của đối tượng. Phương thức có thể là hàm khởi tạo, hàm bình thường hoặc hàm getter/setter.

```cpp
class Person {
private:
    std::string name;
    int age;

public:
    Person(std::string n, int a) : name(n), age(a) {}

    void setName(std::string n) {
        name = n;
    }

    std::string getName() {
        return name;
    }
};
```

3. **Class bên trong class (Nested Class):** Một lớp có thể chứa một lớp khác, gọi là lớp lồng nhau. Lớp bên trong có thể truy cập các thành viên của lớp bên ngoài, và ngược lại, tùy thuộc vào phạm vi truy cập.

```cpp
class Outer {
public:
    class Inner {
    public:
        void display() {
            std::cout << "Inside Inner class" << std::endl;
        }
    };
};
```

4. **Constructor:** Constructor là một phương thức đặc biệt được gọi khi một đối tượng của lớp được tạo ra. Nó được sử dụng để khởi tạo các thuộc tính của đối tượng và cấp phát tài nguyên nếu cần. Constructor có cùng tên với lớp và không có kiểu trả về.

```
class Example {
public:
    int value;

    // Constructor
    Example(int v) : value(v) {
        std::cout << "Constructor called with value: " << value << std::endl;
    }
};
```

5. **Destructor:** Destructor là một phương thức đặc biệt được gọi khi một đối tượng bị hủy. Mục đích của destructor là giải phóng tài nguyên mà đối tượng đã sử dụng. Destructor có cùng tên với lớp nhưng có tiền tố và không có kiểu trả về.

```
class Example {
public:
    int* data;

    // Constructor
    Example(int size) {
        data = new int[size];
        std::cout << "Constructor called, memory allocated" << std::endl;
    }

    // Destructor
    ~Example() {
        delete[] data;
        std::cout << "Destructor called, memory freed" << std::endl;
    }
};
```

6. **Phương thức static (Static Method):** Phương thức static thuộc về lớp chứ không thuộc về một đối tượng cụ thể. Chúng có thể được gọi mà không cần tạo đối tượng của lớp. Phương thức static thường được sử dụng cho các hàm tiện ích hoặc các biến chung của lớp.

```
class MathUtil {
public:
    static int add(int a, int b) {
        return a + b;
    }
};

// Sử dụng phương thức static
int result = MathUtil::add(5, 3);
```

7. **Con trỏ this (This Pointer):** Con trỏ `this` là một con trỏ đặc biệt mà mọi đối tượng có sẵn, trỏ đến đối tượng hiện tại mà phương thức đang làm việc. Nó được sử dụng để truy cập các thuộc tính và phương thức của đối tượng hiện tại.

```cpp
class Person {
private:
    std::string name;

public:
    void setName(std::string name) {
        this->name = name;  // Dùng con trỏ this để phân biệt giữa biến thành viên và tham số
    }
};
```

8. **Tầm vực (Access Specifiers):** Tầm vực xác định quyền truy cập vào các thành viên của lớp. Có ba tầm vực chính:
   - **Public:** Các thành viên được khai báo là public có thể được truy cập từ bất kỳ đâu trong chương trình.
   - **Private:** Các thành viên được khai báo là private chỉ có thể được truy cập từ bên trong lớp.
   - **Protected:** Các thành viên được khai báo là protected có thể được truy cập từ lớp kế thừa và lớp bạn, nhưng không thể truy cập từ bên ngoài lớp.

```cpp
class Example {
public:
    int publicVar;

protected:
    int protectedVar;

private:
    int privateVar;
};
```

9. **Thừa kế (Inheritance):** Thừa kế cho phép một lớp kế thừa các thuộc tính và phương thức từ một lớp khác, gọi là lớp cơ sở (base class). Lớp kế thừa (derived class) có thể mở rộng hoặc thay đổi hành vi của lớp cơ sở.

```cpp
class Animal {
public:
    void eat() {
        std::cout << "Animal eats" << std::endl;
    }
};

class Dog : public Animal {
public:
    void bark() {
        std::cout << "Dog barks" << std::endl;
    }
};

// Sử dụng thừa kế
Dog myDog;
```

```
myDog.eat();  // Gọi phương thức từ lớp cơ sở Animal
myDog.bark(); // Gọi phương thức từ lớp kế thừa Dog
```

10. **Friend Class** friend là một từ khóa được sử dụng để cho phép một lớp hoặc hàm truy cập các thành viên riêng tư (private) hoặc bảo vệ (protected) của một lớp khác. Khi một lớp hoặc hàm được khai báo là friend của một lớp, nó có thể truy cập tất cả các thành viên của lớp đó, ngay cả khi các thành viên đó là private hoặc protected.

```
class A {
    friend class B;  // B có quyền truy cập vào các thành viên private của A

private:
    int secret = 42;
};

class B {
public:
    void revealSecret(const A& a) {
        std::cout << "The secret is: " << a.secret << std::endl;
    }
};
```

3.2 Kiến thức nâng cao

1. **Lớp trừu tượng (Abstract Class):** Lớp trừu tượng là một lớp không thể được khởi tạo trực tiếp và thường chứa ít nhất một phương thức thuần ảo (pure virtual method). Các lớp kế thừa từ lớp trừu tượng phải cài đặt các phương thức thuần ảo để có thể được khởi tạo.

```
class Shape {
public:
    virtual void draw() = 0;  // Phương thức thuần ảo
};

class Circle : public Shape {
public:
    void draw() override {
        std::cout << "Drawing Circle" << std::endl;
    }
};
```

2. **Đa hình với phương thức ảo (Polymorphism with Virtual Functions):** Đa hình cho phép một phương thức trong lớp cơ sở được ghi đè bởi các phương thức trong lớp kế thừa. Sử dụng từ khóa `virtual` trong lớp cơ sở và từ khóa `override` trong lớp kế thừa để hỗ trợ tính năng này.

```
class Animal {
public:
    virtual void speak() {
        std::cout << "Animal speaks" << std::endl;
    }
};

```

```
class Dog : public Animal {
public:
    void speak() override {
        std::cout << "Dog barks" << std::endl;
    }
};

// Sử dụng đa hình
Animal* a = new Dog();
a->speak();  // Output: Dog barks
```

3. **Overload Toán tử (Operator Overloading):** Toán tử có thể được nạp chồng để thực hiện các phép toán tùy chỉnh cho các lớp. Điều này cho phép sử dụng các toán tử như +, -, ==, v.v., với các đối tượng của lớp.

```
class Complex {
private:
    double real;
    double imag;

public:
    Complex(double r, double i) : real(r), imag(i) {}

    Complex operator + (const Complex& other) {
        return Complex(real + other.real, imag + other.imag);
    }

    bool operator == (const Complex& other) {
        return (real == other.real && imag == other.imag);
    }
};
```

4. **Template:** Template cho phép định nghĩa các lớp và hàm với kiểu dữ liệu tùy chỉnh, giúp tăng tính tái sử dụng của mã nguồn. Templates có thể là lớp template (class template) hoặc hàm template (function template).

```
template <typename T>
class Stack {
private:
    std::vector<T> elements;

public:
    void push(const T& element) {
        elements.push_back(element);
    }

    T pop() {
        T elem = elements.back();
        elements.pop_back();
        return elem;
    }
};

```

```
// Sử dụng lớp template
Stack<int> intStack;
intStack.push(1);
intStack.push(2);
```

5. **Shallow Copy và Deep Copy:**
   - **Shallow Copy (Sao chép nông):** Sao chép nông tạo một bản sao của đối tượng, nhưng chỉ sao chép các tham chiếu, không sao chép các đối tượng mà các tham chiếu này trỏ tới. Điều này có thể dẫn đến sự chia sẻ dữ liệu giữa các đối tượng.

```
class Shallow {
private:
    int* data;
public:
    Shallow(int value) {
        data = new int(value);
    }

    ~Shallow() {
        delete data;
    }

    Shallow(const Shallow& other) {
        data = other.data;  // Sao chép con trỏ, không sao chép dữ liệu
    }
};
```

   - **Deep Copy (Sao chép sâu):** Sao chép sâu tạo một bản sao của đối tượng và sao chép cả dữ liệu mà các con trỏ của đối tượng trỏ tới, tránh sự chia sẻ dữ liệu không mong muốn.

```
class Deep {
private:
    int* data;
public:
    Deep(int value) {
        data = new int(value);
    }

    ~Deep() {
        delete data;
    }

    Deep(const Deep& other) {
        data = new int(*other.data);  // Sao chép dữ liệu, không chỉ con trỏ
    }
};
```

4 Kiến thức cơ bản về Array List

Lớp `ArrayList` là một cấu trúc dữ liệu động được sử dụng để lưu trữ danh sách các phần tử trong mảng động. Lớp này cho phép mở rộng mảng khi cần và cung cấp các phương thức để thao tác với danh sách.

Các Thành Phần Chính

- **data:** Con trỏ đến mảng động (`T *data`). Đây là nơi lưu trữ các phần tử của danh sách.
- **capacity:** Biến lưu trữ kích thước hiện tại của mảng động.
- **count:** Biến lưu trữ số lượng phần tử hiện tại trong danh sách.

Các Phương Thức Cơ Bản

- **Constructor:** Khởi tạo danh sách với kích thước mặc định hoặc theo yêu cầu.
- **Destructor:** Giải phóng bộ nhớ của mảng động để tránh rò rỉ bộ nhớ.
- **Copy Constructor và Assignment Operator:** Sao chép một đối tượng `ArrayList` sang một đối tượng khác.
- **Thêm và Xóa Phần Tử:** Thêm hoặc xóa các phần tử khỏi danh sách, điều chỉnh `count` và kích thước mảng nếu cần.
- **Truy Xuất Phần Tử:** Truy cập phần tử tại một vị trí cụ thể trong danh sách.

Ví Dụ Minh Họa

```
template <typename T>
class ArrayList {
protected:
    T* data;          // Mảng động để lưu trữ các phần tử
    int capacity;     // Kích thước của mảng động
    int count;        // Số lượng phần tử hiện tại

    void resize(int newCapacity) {
        T* newData = new T[newCapacity];
        for (int i = 0; i < count; ++i) {
            newData[i] = data[i];
        }
        delete[] data;
        data = newData;
        capacity = newCapacity;
    }

public:
    // Constructor
    ArrayList(int initialCapacity = 10) : capacity(initialCapacity), count(0) {
        data = new T[capacity];
    }

    // Destructor
    ~ArrayList() {
        delete[] data;
    }

    // Copy Constructor
```

```
30      ArrayList(const ArrayList& other) : capacity(other.capacity), count(other.count)
        ↪ {
31          data = new T[capacity];
32          for (int i = 0; i < count; ++i) {
33              data[i] = other.data[i];
34          }
35      }
36
37      // Assignment Operator
38      ArrayList& operator=(const ArrayList& other) {
39          if (this != &other) {
40              delete[] data;
41              capacity = other.capacity;
42              count = other.count;
43              data = new T[capacity];
44              for (int i = 0; i < count; ++i) {
45                  data[i] = other.data[i];
46              }
47          }
48          return *this;
49      }
50
51      // Thêm phần tử
52      void add(const T& item) {
53          if (count == capacity) {
54              resize(capacity * 2);
55          }
56          data[count++] = item;
57      }
58
59      // Xóa phần tử
60      void remove(int index) {
61          if (index < 0 || index >= count) {
62              throw std::out_of_range("Index out of bounds");
63          }
64          for (int i = index; i < count - 1; ++i) {
65              data[i] = data[i + 1];
66          }
67          --count;
68      }
69
70      // Truy xuất phần tử
71      T& get(int index) {
72          if (index < 0 || index >= count) {
73              throw std::out_of_range("Index out of bounds");
74          }
75          return data[index];
76      }
77
78      // Số lượng phần tử hiện tại
79      int size() const {
80          return count;
81      }
82  };
```

Giải Thích

- **resize:** Khi mảng đầy, phương thức `resize` mở rộng kích thước của mảng để chứa thêm phần tử.
- **Copy Constructor và Assignment Operator:** Đảm bảo sao chép đúng cách các đối tượng `ArrayList`, bao gồm sao chép dữ liệu và cấp phát bộ nhớ.
- **Exception Handling:** Xử lý lỗi như chỉ số ngoài phạm vi để đảm bảo tính ổn định của chương trình.

5 Kiến thức cơ bản về Doubly Linked List

Lớp `DLinkedList` đại diện cho một danh sách liên kết đôi, nơi mỗi nút có hai con trỏ: một trỏ đến nút kế tiếp và một trỏ đến nút trước đó. Lớp này cung cấp các phương thức để thao tác với danh sách liên kết đôi, bao gồm thêm, xóa và truy cập các phần tử.

Các Thành Phần Chính

- **head:** Con trỏ đến nút đầu tiên trong danh sách. Nút này không chứa dữ liệu người dùng mà chỉ là điểm bắt đầu của danh sách.
- **tail:** Con trỏ đến nút cuối cùng trong danh sách. Nút này không chứa dữ liệu người dùng mà chỉ là điểm kết thúc của danh sách.
- **count:** Biến lưu trữ số lượng phần tử hiện tại trong danh sách.

Các Phương Thức Cơ Bản

- **Constructor:** Khởi tạo danh sách liên kết đôi với các con trỏ `head` và `tail` được đặt thành `nullptr`, và `count` thành 0.
- **Destructor:** Giải phóng bộ nhớ của các nút trong danh sách để tránh rò rỉ bộ nhớ.
- **Copy Constructor và Assignment Operator:** Sao chép một đối tượng `DLinkedList` sang một đối tượng khác.
- **Thêm và Xóa Phần Tử:** Thêm hoặc xóa các phần tử khỏi danh sách, điều chỉnh `count` và các con trỏ `head` và `tail` nếu cần.
- **Truy Xuất Phần Tử:** Truy cập phần tử tại một vị trí cụ thể trong danh sách.

Ví Dụ Minh Họa

```
template <typename T>
class DLinkedList {
public:
    class Node {
    public:
        T data;
        Node* next;
        Node* prev;
    };
protected:
    Node<T>* head;  // Con trỏ đến nút đầu tiên
    Node<T>* tail;  // Con trỏ đến nút cuối cùng
    int count;      // Số lượng phần tử hiện tại

public:
    // Constructor
    DLinkedList() : head(nullptr), tail(nullptr), count(0) {}

    // Destructor
    ~DLinkedList() {
        Node<T>* current = head;
        while (current != nullptr) {
            Node<T>* next = current->next;
            delete current;
            current = next;
```

```
        }
    }

    // Copy Constructor
    DLinkedList(const DLinkedList& other) : head(nullptr), tail(nullptr), count(0) {
        Node<T>* current = other.head;
        while (current != nullptr) {
            add(current->data);  // Giả sử có phương thức add()
            current = current->next;
        }
    }

    // Assignment Operator
    DLinkedList& operator=(const DLinkedList& other) {
        if (this != &other) {
            // Xóa tài nguyên cũ
            while (head != nullptr) {
                Node<T>* temp = head;
                head = head->next;
                delete temp;
            }
            count = 0;

            // Sao chép tài nguyên mới
            Node<T>* current = other.head;
            while (current != nullptr) {
                add(current->data);  // Giả sử có phương thức add()
                current = current->next;
            }
        }
        return *this;
    }

    // Thêm phần tử vào danh sách
    void add(const T& item) {
        Node<T>* newNode = new Node<T>{item, nullptr, tail};
        if (tail != nullptr) {
            tail->next = newNode;
        }
        tail = newNode;
        if (head == nullptr) {
            head = newNode;
        }
        ++count;
    }

    // Xóa phần tử tại vị trí cụ thể
    void remove(int index) {
        if (index < 0 || index >= count) {
            throw std::out_of_range("Index out of bounds");
        }

        Node<T>* current = head;
        for (int i = 0; i < index; ++i) {
            current = current->next;
        }
```

```
82
83          if (current->prev != nullptr) {
84              current->prev->next = current->next;
85          }
86          if (current->next != nullptr) {
87              current->next->prev = current->prev;
88          }
89          if (current == head) {
90              head = current->next;
91          }
92          if (current == tail) {
93              tail = current->prev;
94          }
95
96          delete current;
97          --count;
98      }
99
100     // Truy xuất phần tử tại vị trí cụ thể
101     T get(int index) const {
102         if (index < 0 || index >= count) {
103             throw std::out_of_range("Index out of bounds");
104         }
105         Node<T>* current = head;
106         for (int i = 0; i < index; ++i) {
107             current = current->next;
108         }
109         return current->data;
110     }
111
112     // Số lượng phần tử hiện tại
113     int size() const {
114         return count;
115     }
116 };
```

Giải Thích

- **add:** Thêm phần tử vào cuối danh sách và cập nhật con trỏ `tail`.
- **remove:** Xóa phần tử tại một vị trí cụ thể và cập nhật các con trỏ liên kết.
- **Copy Constructor và Assignment Operator:** Đảm bảo sao chép đúng cách danh sách liên kết đôi, bao gồm sao chép các nút và điều chỉnh liên kết.
- **Exception Handling:** Xử lý lỗi như chỉ số ngoài phạm vi để đảm bảo tính ổn định của chương trình.

6 Kiến thức cơ bản về Iterator

Iterator là đối tượng dùng để duyệt qua các phần tử của một container như mảng, danh sách liên kết, và các cấu trúc dữ liệu phức tạp khác. Nó cung cấp một giao diện đồng nhất để truy cập các phần tử và thực hiện các thao tác như tăng chỉ số và giải tham chiếu.

Các Thành Phần Chính

- **Constructor:** Khởi tạo iterator, thường bằng cách đặt nó trỏ đến một phần tử cụ thể trong container.
- **Assignment Operator:** Cho phép gán giá trị của iterator từ một iterator khác.
- **Remove Method:** Phương thức xóa phần tử mà iterator đang trỏ đến (có thể yêu cầu tham chiếu đến container).
- **Dereference Operator (&operator*):** Trả về tham chiếu đến phần tử mà iterator đang trỏ đến.
- **Prefix Increment Operator (++i):** Di chuyển iterator đến phần tử tiếp theo và trả về iterator đã tăng.
- **Postfix Increment Operator (i++):** Di chuyển iterator đến phần tử tiếp theo nhưng trả về bản sao của iterator trước khi tăng.

Ví Dụ Minh Họa

```cpp
template <typename T>
class Iterator {
private:
    T* ptr;  // Con trỏ đến phần tử hiện tại

public:
    // Constructor
    Iterator(T* p = nullptr) : ptr(p) {}

    // Assignment Operator
    Iterator& operator=(const Iterator& other) {
        if (this != &other) {
            ptr = other.ptr;
        }
        return *this;
    }

    // Remove Method
    void remove() {
        // Phương thức xóa phần tử (cần tham chiếu đến container thực tế)
        // container.remove(ptr);
    }

    // Dereference Operator
    T& operator*() const {
        return *ptr;
    }

    // Prefix Increment Operator
    Iterator& operator++() {
        ++ptr;
        return *this;
```

```
    }

    // Postfix Increment Operator
    Iterator operator++(int) {
        Iterator temp = *this;
        ++ptr;
        return temp;
    }
};
```

Giải Thích

- **Constructor:** Khởi tạo iterator với con trỏ trỏ đến phần tử cụ thể trong container.
- **Assignment Operator:** Cho phép gán giá trị của iterator từ iterator khác, hữu ích trong các phép sao chép.
- **Remove Method:** Có thể xóa phần tử mà iterator đang trỏ đến; thường yêu cầu quyền truy cập vào container.
- **Dereference Operator (&operator*):** Trả về tham chiếu đến giá trị mà iterator đang trỏ đến, cho phép truy cập và sửa đổi giá trị đó.
- **Prefix Increment Operator (++i):** Di chuyển iterator đến phần tử tiếp theo và trả về iterator đã được tăng.
- **Postfix Increment Operator (i++):** Di chuyển iterator đến phần tử tiếp theo và trả về bản sao của iterator trước khi tăng, giữ lại giá trị cũ của iterator.